High Performance Publish / Subscribe Middleware for Wireless Sensor Network

Steven Lai

Ph.D

The Hong Kong Polytechnic University

2013

The Hong Kong Polytechnic University

Department of Computing

High Performance Publish / Subscribe Middleware for Wireless Sensor Network

Author:

Steven Lai

Supervisor:

Dr. Jiannong Cao

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

September 2012

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

_____ **(Signed)**

Lai Yi

_____ **(Name of student)**

Dedication

To my supervisor Prof. Cao and my colleagues in IMCL for their continuous help on my work. To my family and my friends, Amber, Alice, Engel, Kim, etc. who have supported me during my most difficult time.

Abstract

Many WSN applications such as intelligent transportation systems, smart building and intelligent car park are event-based in the sense that these applications require specifying, detecting and delivering events which are of users' interest and may trigger necessary actions. However, developing event-based applications is challenging because of two reasons. First, it is difficult to specify and detect composite events. Second, different applications may have different requirements for event processing such as communication reliability, network life time or delay bound.

In general, an event-based system may have primitive or composite events. Primitive events, such as threshold of temperature, may be detected by single sensor node. Composite events, on the other hand, is defined by the relations among primitive events and must be detected by multiple sensor nodes cooperatively. Although there are several works on providing event-based services in WSN, most of them can only deal with primitive event types but cannot handle composite events very well. In this research work, we introduce PSWare, a type-based publish / subscribe middleware framework for WSN that supports composite events. PSWare provides a declarative event definition language for the users to specify the composite events. It also facilitates easy configuration of different event processing algorithms and integrate them into the system at run time.

Based on PSWare, we present TED (Type-based Event Detection), a novel distributed composite event detection algorithm. The essential idea of TED is event fusion, where some sensor nodes are selected as fusion points and component events are fused for the detection of a high level event. Event fusion with minimum energy cost is an NP-complete problem. Therefore, TED uses a number of heuristics with bounded performance.

We use PSWare to develop applications in certain domains such as Intelligent Transportation Systems (ITS) and Structural Health Monitoring (SHM). We design algorithms so that PSWare is customized to work well in these application domains. In particular, we design a clustering algorithm for PSWare to use in SHM. We formulate the clustering problem and found it to be NP-complete so we propose heuristic centralized and distributed algorithms.

We evaluate PSWare from different aspects. We evaluate TED through analysis and extensive simulation. Both analytical and simulation results show TED can save energy in event-based applications where primitive events occur in a higher frequency than composite events. Then we carry out some real world experiments using PSWare. The results show that PSWare can offer reasonably simple API for the application developers to use while TED and our clustering algorithm can improve the underlying event detection performance. Compared with opportunistic approaches to event detection in these real applications, PSWare can reduce 40 - 50 % of the energy cost.

Publications

- Steven Lai, Jiannong Cao, and Xiaopeng Fan. Ted: Efficient type-based composite event detection for wireless sensor network. In *Proceedings of 7th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'11)*, June 2011

- Steven Lai, Jiannong Cao, and Yuan Zheng. Psware: A publish / subscribe middleware supporting composite event in wireless sensor network. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications (PerSeNS'09)*, pages 1–6, March 2009

- Yi Lai, Yuan Zheng, and Jiannong Cao. Protocols for traffic safety using wireless sensor network. In *Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'07)*, pages 37–48, June 2007

- Xuefeng Liu, Jiannong Cao, Steven Lai, Chao Yang, Hejun Wu, and Youlin Xu. Energy efficient clustering for wsn-based structural health monitoring. In *Proceedings of 30th IEEE International Conference on Computer Communications (INFOCOM'11)*, April 2011

- Xuefeng Liu, Jiannong Cao, Md. Zakirul Alam Bhuiyan, Steven Lai, Hejun Wu, and Guojun Wang. Fault tolerant wsn-based structural health monitoring. In *Proceedings of 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'11)*, June 2011

- Vaskar Raychoudhury, Jiannong Cao, Weigang Wu, and Steven Lai. K-directory community: Reliable service discovery in manet. *Journal of Pervasive and Mobile Computing (JPMC)*, 7(1), February 2011

- Jiannong Cao, Hejun Wu, Xuefeng Liu, and Yi Lai. isensnet: an infrastructure for research and development in wireless sensor networks. *Frontiers of Computer Science in China*, 4(3):339–353, 2010

Acknowledgements

This work is supported by Hong Kong RGC under CERG grant PolyU5102/08E and the Hong Kong Polytechnic University under Niche Area Project grant 1-BB6C.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Development in wireless communication and electronics has made it possible to create low-cost, low-power wireless sensor nodes. Each sensor node usually contains a wireless transceiver, a micro processing unit and a number of sensors. The sensor nodes can collect data and do some simple processing locally, and can communicate with each other to form an ad hoc wireless sensor network (WSN) [3]. A WSN is usually self-organized and self-maintained without any human intervention. Wireless sensor networks have been used in various application areas such as smart building [76], wild environment monitoring [80], intelligent transportation systems [54], battle surveillance [36] and healthcare [74].

While WSN has a wide range of applications, programming sensor networks is a challenging because different from programming in the traditional environment. WSN imposes a lot of constraints such as limited compu-

tational power, less memory and unreliable communication. Application developers need to not only understand the requirements for the specific application domain but also take into consideration of the characteristics of WSN. In this research work, we propose PSWare, a publish/subscribe (pub/-sub) middleware for WSN which eases the development of WSN applications. Our middleware uses a pub/sub programming paradigm for the application programmer to subscribe application specific events. It provides an easy-to-use Event Definition Language (EDL) to allow the application developers to define composite events while uses a flexible architecture so that different domain-specific event processing algorithms can be easily integrated into the middleware.

1.1.1 Motivation

Despite the large variety of WSN applications, many of them are essentially event-based. In applications such as intelligent transportation systems [54], smart buildings [76] and healthcare [74], the events sensor nodes detect events which reflect the environmental changes and the systems respond to these events accordingly. Events may be primitive or composite. Primitive events (e.g. when the temperature exceeds certain threshold) can be detected by a single sensor node without having to cooperate with others. On the other hand, *composite events* consist of multiple primitive events. They reflect a serial of environmental changes with spatial and temporal relations among them and must be detected collaboratively by different sensor nodes.

Because of the common requirements and challenges for composite event

subscription and detection, it is more desirable to have a generic middleware layer to handle composite events instead of reinventing the wheel and implementing application-specific event processing mechanisms. In addition, the middleware should be flexible enough so that different event processing algorithms for different application domains can be easily integrated. In summary, the middleware should achieve the following design goals:

1. *Event abstraction*: since composite events are collaboratively processed by different sensor nodes in the network, they introduce extra complexity because of the resource constraints and unreliable communication. A middleware framework providing high level event abstraction is needed to ease the application development.

2. *Re-usability and flexibility*: different applications may share certain common modules during event processing. They may also have other different modules. A middleware framework can help so that common modules may be used and different modules may be replaced without affecting others.

1.1.2 Issues

In this research, we address the essential issues of designing and developing PSWare. On the highest level, an event description language (EDL) is provided to allow users to describe composite event relationships. On the lowest level, a runtime environment is necessary on each sensor nodes so that they could understand and execute the event processing algorithms translated from the high-level EDL. More specifically, we need to address the following

research/engineering issues to make our middleware work.

- *Event definition language*: we need to provide an event description language which is powerful yet easy to use.
- *Event definition language compiler*: a compiler is essential to translate the programs written in our high-level language into a low-level language understandable by the sensor nodes. The compiler must be smart enough to extract the semantic meanings from the program and do some optimization.
- *Runtime environment support*: the middleware running on each sensor node should provide a runtime environment to execute the compiled programs.
- *Subscription propagation*: after the program written in EDL is compiled, the subscription should be intelligently disseminated into the network. If the subscription is too big, it may need to be divided into small ones. Furthermore, only related sensor nodes need to be updated.
- *Event detection*: we need to design efficient protocol for the sensor nodes to cooperate with each other to detect composite events.
- *Event delivery*: after the subscribed events have been detected, we need energy-efficient routing protocols so that the detected events will be delivered at the minimum cost.

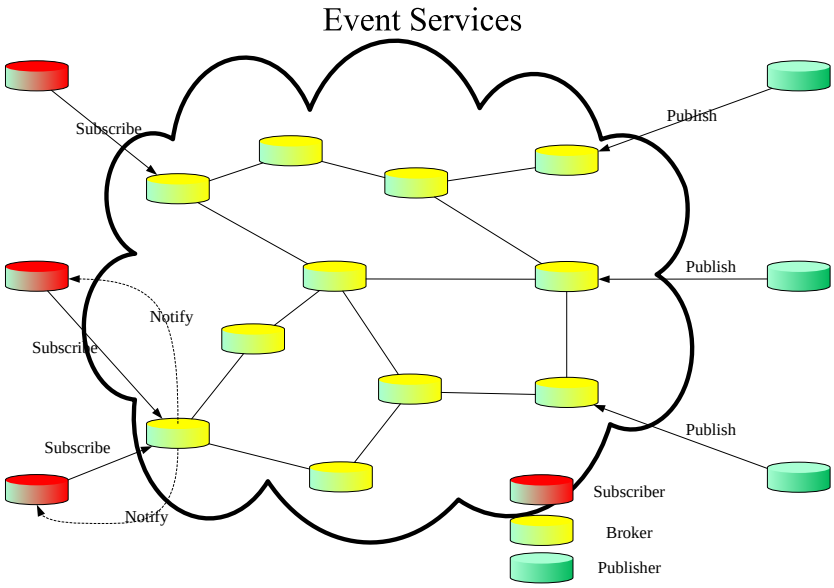


Figure 1.1: A conceptual model for distributed pub/sub systems

1.2 Background

In this section, we briefly overview the background of pub/sub systems and WSN middleware.

1.2.1 Publish / Subscribe Systems

The publish/subscribe (pub/sub) communication paradigm has received an increasing attention because of its loosely coupled nature. Conceptually, a distributed pub/sub system model can be illustrated in Figure 1.1.

An essential part of a pub/sub system is the event service which is responsible for managing subscriptions and detecting the events based on the subscriptions. Once an event is detected, the corresponding subscribers interested in that event will be notified. The tasks required by the event service

are usually done by brokers. In a distributed pub/sub system, there can be many brokers in the network and they usually cooperate with each other in order to provide the event service. Apart from the brokers, a pub/sub system usually include other entities such as subscribers and publishers. Subscribers register their interest in events by calling a subscribe() operation to the event service. To fire an event, a publisher typically calls a publish() operation. The event service notifies the subscribers by calling notify() to all relevant subscribers.

Compared with Traditional interaction paradigms such as message passing, RPC [8], notification [28], shared spaces [14] and message queuing [5], pub/sub paradigm allows decoupling between publishers and subscribers in three dimensions [23].

- *Space decoupling*: the interacting parties do not need to know each other. The publishers publish events through an event service and the subscribers get these events indirectly from the event service. The publishers do not usually have any reference to these subscribers; neither do they know how many of these subscribers are participating in the interaction. Similarly, subscribers do usually not have any reference to the publishers; neither do they know how many of these publishers are participating in the interaction.

- *Time decoupling*: the interacting parties do not need to be actively participating in the interaction at the same time. In particular, the publisher might publish some events while the subscriber is disconnected, and conversely, the subscriber might get notified about the occurrence

of some event while the original publisher of the event is disconnected.

- *Flow decoupling*: publishers are not blocked while producing events and subscribers can get notified about the occurrence of some event while performing some concurrent activity (through a callback), i.e., subscribers do not need to pull for events in a synchronous manner. In short, message production and consumption do not happen in the main flow of control of the publisher or subscriber.

Because of the decoupling features of pub/sub paradigm, developing applications based on pub/sub paradigm will become easier easy. A few pub-/sub systems such as Siena [15], Elvin [97], Gryphon [4] and Jedi [20] have already been implemented in distributed networks.

1.2.2 WSN Middleware

WSN offers an opportunity for a wide range of applications. The sensor nodes are low cost, low power and easily deployable. When combined, they offer numerous advantages over traditional networks, such as a large-scale flexible architecture, high-resolution sensed data and application adaptive mechanisms. However, due to their tight integration to the physical world, sensor networks pose considerable impediments and make application development nontrivial. Using middleware to bridge the gap between applications and low-level constructs is an attractive approach to resolving many WSN issues and easing the application development. Consequently, a lot of WSN middlewares have been proposed and they provide different kinds of programming abstraction.

Conventionally, middleware is a software layer between operating system and applications. It abstracts the underlying details provided by OS and provides a high level API for applications. The purpose of middleware is usually for easier application development because the application developers can make use of the easier and higher level API instead of the ones provided by OS. WSN middleware is similar in the sense the middleware also sits between the operating system and the application. Moreover, WSN middleware allows the application to view the network as a whole and therefore, program it without being aware of where the data is stored and transmitted. A conceptual model for a WSN middleware is illustrated in Figure 1.2.

1.2.3 Events and Subscriptions

In our pub/sub model, each event is a list of attributes which are the actual data obtained from the sensor network [42]. Applications can subscribe events with subscriptions which contain event types [23] and event filters [15]. Events can have relations among each other. The relations are expressed as event operators.

Events together with their relations can be represented as a directed acyclic graph where each node represents an event and the each edge represents a relation. If the relations are all binary relations, each node in the graph can have an indegree of either 0 or 2. Figure 1.3a shows an example of such a graph. The nodes with 0 indegree represent primitive events. The rest of the nodes represent composite events. The event which has 0 outdegree is the subscribed event.

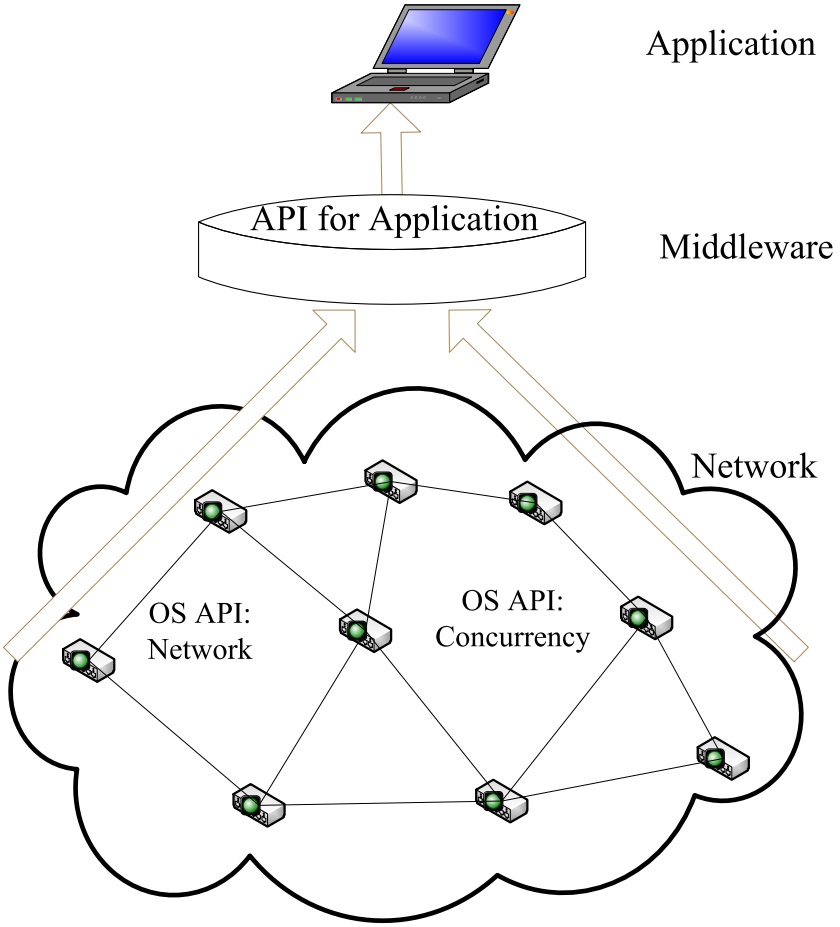
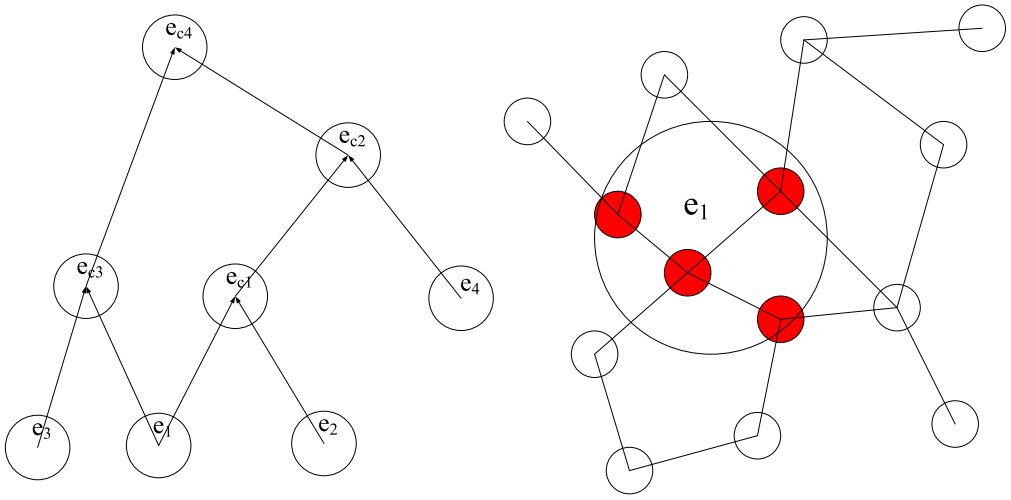


Figure 1.2: WSN middleware



(a) Event hierarchy (b) Event detection through aggrgation

Figure 1.3: Event definition and detection

Once the subscriptions are defined, it will be disseminated into the network so that sensor nodes can start to collect data and detect events. The sensor nodes collect data in rounds. In each round, the collected data will be matched against the subscription. If an event is found to match the subscribed event type, it will be delivered to the sink to notify the subscriber.

1.2.4 Event Detection and Detection Cost

Events can be detected with different strategies based on their types and relations. In this work, we consider the detection cost as message cost. The simplest type of event is the primitive event. Existing work may not have a clear definition on primitive events in the sense that such event may be detected by a single node or by multiple nodes. In the case of multiple nodes, the events are usually detected in the form of data aggregation with aggregates such as average, max or sum. For the sake of discussion, in this

work, we consider primitive events as only those that can be detected by single sensor node. For events which involve data aggregation, we consider them as composite events as discussed shortly. Since such detection does not involve collaboration among nodes, the communication cost for detecting primitive events is always zero.

The second type of event is composite event. Composite events are detected based on a number of sub-events. These sub-events may be primitive or composite events. Apparently, nodes must collaborate in order to detect composite events. Therefore, the communication cost for detecting a composite event comprise of the communication cost for each sub-event and the additional cost for forwarding these sub-events to a node for detection. Figure 1.3b shows an example of a composite event that merely aggregates the primitive events. Since four primitive events are used, the communication cost for detecting the event is 4.

Apart from data aggregation such as sum, max or average, a composite event may require more complicated spatial or temporal relations among its sub-events. A sensor node then needs to be selected to first save the detected sub-events. As shown in Figure 1.4 if a composite event is detected based on two or more sub-events e_1 and e_2, then the detected e_1 and e_2 will first be forwarded to a node in their topological center. In this paper, we refer to such node as event fusion point. The communication cost for detecting the composite event will be the cost for detecting each individual sub-events together with the cost for delivering the event detection results to the event fusion point.

Sometimes the sub-events may not always happen. In this case, we don't

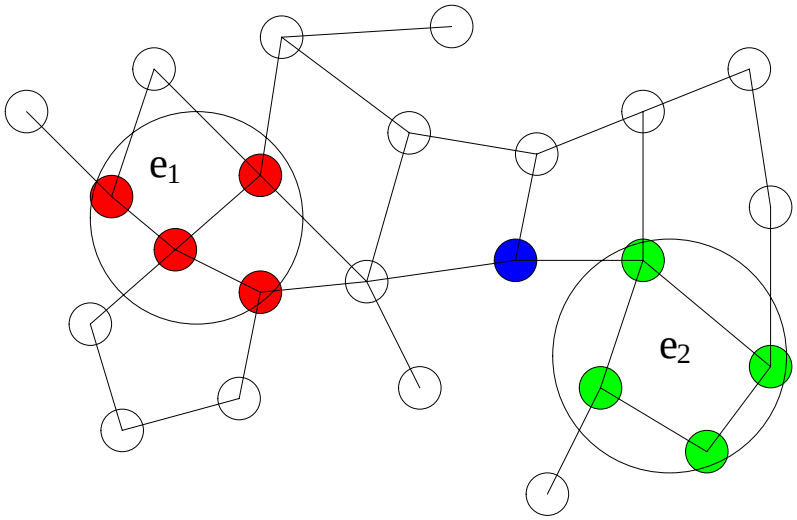


Figure 1.4: Composite event detection through event fusion point

need to always send the results to the event fusion point. Instead, we can forward only the detected events in order to further save the energy. For example, as shown in Figure 1.5, if we have two events e_1 and e_2 which have a relation such that the composite event happens only when both events have been successfully detected. If e_2 needs to be monitored at a higher rate than e_1 and the probability that e_2 will really occur is also higher than that of e_1. Then we can further move the fusion point closer to e_2 so that the nodes do not need to start detecting less frequent e_2 if e_1 has not been detected. In this way, we can further reduce the communication cost because the nodes do not need to spend energy continuously trying to detect e_2.

1.2.5 Composite Event-based Applications

We consider several applications that motivated the development of PSWare. The first one is indoor monitoring for fire detection. When a fire occurs,

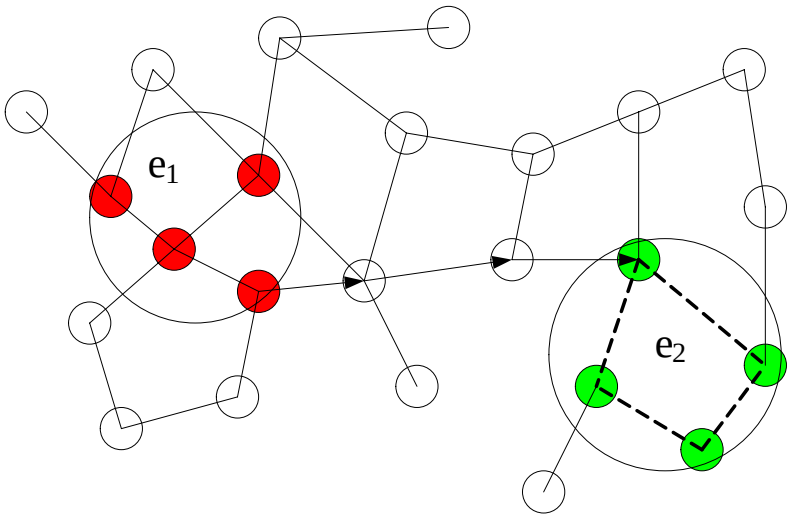


Figure 1.5: Composite event detection using event probability

usually more than one sensor node can detect changes in sensory data such as temperature and light. In addition, when the rescue team comes, they not only need to know the location of the fire but also its spreading speed and direction. This requires detecting changing patterns of the events. In such applications, events will be defined as location of the sensors along with the changes of temperature and light over the time.

The second application is WSN-based structural health monitoring (SHM). The objective of such system is to detect damages on structures such as buildings and bridges if they occur. Event detection is important in these applications because the SHM sensors will introduce high energy consumption during damage detection. It is therefore more desirable to wake them up only upon the occurrence of certain events [49]. One of such example is to start the sensors when certain vibration pattern has been detected on the building to be monitored. In such applications, events may be defined as

vibration patterns over the structure.

Our last application is intelligent transportation systems (ITS). ITS may include a wide array of applications such as traffic light control, road enforcement and congestion information. Many of such applications need the vehicle information. In these applications, we may define vehicle related events as primitive events. Then based on those events, different applications may define different composite events which are important to the services being provided.

From these applications, we can see event detection is an essential part. Moreover, for each application domain, it usually have some basic events such as car events for ITS and temperature event for fire monitoring. Then based on these basic events, different composite events may be defined to further meet the requirements of these applications.

1.3 Thesis Outline

The remaining of the thesis is organized in the following way. First, we review the literature in Chapter 2. We first identify the most relevant topics to PSWare. There are mainly four related areas: WSN middleware, WSN macro-programming, event definition and data aggregation for WSN. For each of these areas, we categorize the existing works and discuss the features based on these categories. Before getting into the technical details for these works, we first try to illustrate their high-level ideas with diagrams.

In Chapter 3, we describe the design of our middleware. Our design uses a top-down approach. We first overview the entire system design and list

the most important design issues including compiler design, event detection algorithm and sensor placement and clustering. Then, we start from the highest level of our middleware - the EDL compiler that directly interact with the user through subscriptions. After the EDL compiler, we describe the design of our runtime environment that supports our event subscription and detection. Then we describe our event detection framework which can be used to easily integrate different types of event detection algorithms.

We go into our first research issue - the composite event detection problem in Chapter 4. We formulate the problem and describe our solution in details. In addition to the main algorithm, we discuss a sub-problem, the fusion point deployment problem. We discuss our second research issue in Chapter 5. This issue came from the structural health monitoring (SHM) application in WSN. We formulate the problem and propose the corresponding solutions.

We summarize our implementation details in Chapter 6. We discuss how different real world applications such as ITS or SHM, can be developed by into PSWare.

We describe our system evaluation in Chapter 7. We first evaluate our generic event detection algorithm, TED, through analysis and simulation. Then we integrate TED into PSWare and perform evaluation through experiments.

In Chapter 8, we conclude our work and discuss the possible future directions.

A Thesis Submitted by Steven Lai

Chapter 2

Literature Review

2.1 Overview

There are many issues related to designing PSWare including:

- Programming abstraction: since the purpose of middleware is to help develop applications, an appropriate layer of abstraction is needed so that the application developers do not have to worry too much about the underlying details.

- Middleware design: the middleware is designed with such an objective in mind that different types of event detection algorithms may be easily integrated into the system. This requires a flexible and extensible architectural design.

- Optimized event detection: an efficient event detection algorithm is the heart of the middleware because the whole system relies on it for efficient event processing.

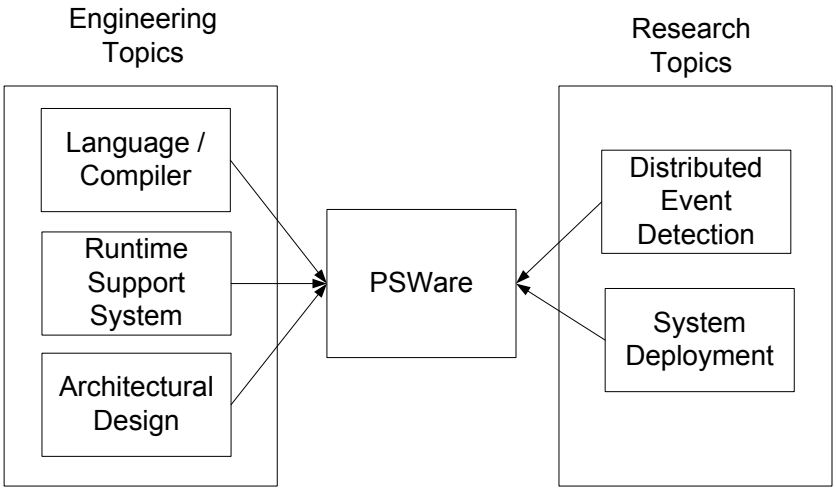


Figure 2.1: Topics related to PSWare

In summary, the issues related to PSWare design can be illustrated in Figure 2.1. From the engineering perspective, we need to provide a high level event definition language to the user. By doing that, we need a compiler for such event definition language. After the high level event definition has been compiled, we need a runtime environment which will act as a glue layer between the compiled codes and the actual event definition algorithm. From the research perspective, the event detection problem mainly involves two aspects - distributed event processing and sensor nodes placement. The distributed processing part emphasizes on optimization so that the events may be detected with minimum energy cost. The placement part studies how the sensor nodes may be efficiently deployed in the network to help achieving the optimization goal.

In this chapter, we study the existing literatures and work on these topics.

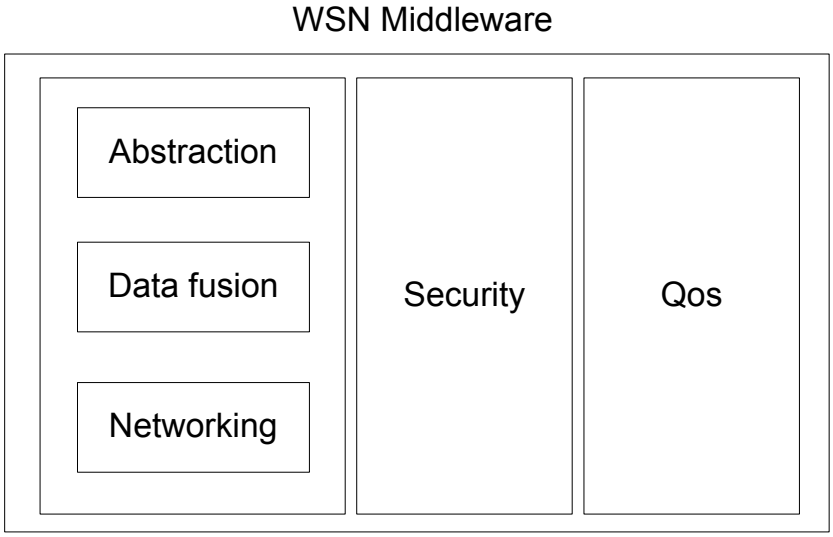


Figure 2.2: Issues in designing WSN middleware

2.2 WSN Middleware

A lot of works on middleware for WSN have been proposed. Different works provide different features to the upper application layer. In this section, we first describe the issues and challenges in designing middleware for WSN. Then, we survey some of the most important works in the area of middleware for WSN. Finally, we summarize the exiting works and analyze some remaining yet unsolved issues.

2.2.1 Issues in Designing WSN Middleware

WSN brings a lot of benefit as well as some challenges. To design and implement an effective and efficient middleware for WSN, theses issues need to be addressed. A few surveys [96, 84, 45, 39] have been done to summarize the existing works. In this section, we will select some key issues regarding to the challenges for designing WSN middleware. The overview of the issues

is illustrated in Figure 2.2. On the left-hand side, abstraction support, data processing and networking may be handled at different software layers. On the right-hand side, QoS and security are usually handled across different software layers.

Abstraction Support: WSN usually consists of large number of heterogeneous sensors. These sensors may be developed by different vendors and may have different capability, precision and platform. Therefore, the middleware should hide the underlying platform details and provide a high level programming paradigm for the application programmers. Ideally, a good programming paradigm allows programming the sensor network as a single 'virtual' entity, rather than individual nodes [96].

Resource Constraints: usually each sensor node in the network has limited resources like energy, computing power, memory and communication bandwidth. Hence, middleware components have to be lightweight to fit these constraints. Middleware should also provide support to dynamically adapt performance and resource consumption to the varying needs of the application, for example, by enabling dynamic tradeoffs between output quality and resource consumption. In addition, multiple applications may run on the same network so the middleware needs to fairly allocate resources between different applications.

Scalability: scalability is defined as follows. If an application grows, the network should be flexible enough to allow this growth anywhere and anytime without affecting network performance [39]. As sensor nodes are small and cheaper, sensor networks are usually deployed in very large scales. As a result, middleware should support mechanisms for self-configuration and self-

maintenance of collections of sensor nodes.

Network Dynamics: ad hoc networks of sensor nodes may exhibit a highly dynamic network topology because of node mobility, environmental obstructions or hardware failures. Middleware should support the robust operation of sensor networks by coping with the changing network environment.

Data Fusion: most sensor network applications involve nodes that contain redundant data and are located in a specific local region. This property makes it possible for in-network data fusion of data from different sources. Data aggregation saves energy by reducing the total amount of data needs to be transferred.

Cross-layer Design: different from traditional middleware which can support a wide variety of applications, WSN middleware cannot be generalized in this way due to limited resource availability. Therefore, most works on WSN middleware include mechanisms for including the application knowledge in the middleware. This lets developers may application requirements to network parameters, which enable them to fine-tune network monitoring. Much existing middleware is coupled to a specific type of applications. However, middleware is intended to support a wide range of applications so developers must explore the trade-offs between the degree of application specificity and middleware generality [39].

Security: WSN middleware need to handle security issues in data processing and data communication. Since sensor networks are often deployed for sensitive applications like military surveillance, patient monitoring, forecasting systems, etc, data collected and distributed by these sensors will have to be genuine and authentic [84]. However, due to limited resource availability,

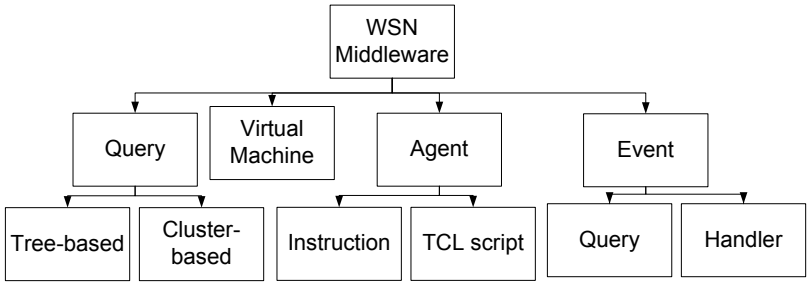


Figure 2.3: Category of existing WSN middleware

existing security algorithms may not be suitable for WSN so new security schemes need be developed to meet the security requirements of WSN.

QoS Support: in WSN, we can view QoS from two perspectives, application and network. The former refers to QoS parameters specific to the application, such as sensor node measurement, deployment, coverage and number of active sensor nodes. The latter refers to how the supporting communication network can meet application needs while efficiently using network resources such as bandwidth and power consumption. Sometimes application-specific QoS requirements may conflict with network QoS requirements. For example, the application may always want to have more sensor nodes being active at the same time but the network, on the other hand, wants to let most of the sensors sleep. Therefore, it is up to middleware to balance the QoS requirements from both application and network perspective.

We categorize the existing WSN middleware according to the programming abstraction provided. Programming abstraction is the most direct way for developers to interact with middleware. Figure 2.3 shows the category of existing middleware.

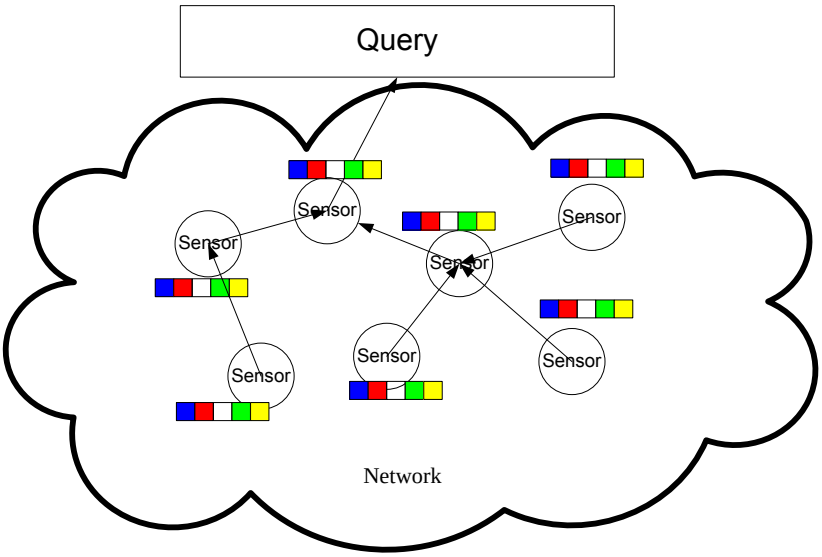


Figure 2.4: Query-based middleware for WSN

2.2.2 Query-based Middleware

Query-based approach provides a SQL-like query language to higher level. As shown in Figure 2.4, high level applications interact with the network using queries. To allow faster data delivery, the underlying network may group the data according to their types. The nodes may form a tree or clusters to further speed up the query processing.

The earliest work falling into this category is COUGAR [10]. On the highest level, it uses SQL like query language. Sensors are modeled as abstract data types (ADT) in an object oriented fashion so the signal processing functions can be represented as a method of a type. An initial version of Cougar has been implemented on the actual sensor.

Some work such as TinyDB [79] uses a query-based approach with a couple of distinct language features such as event-based queries and aggregation. Internally, it defines some metadata for query optimization. A few power-

based query optimization techniques were introduced such as power-based ordering and event query batching. For query dissemination, a semantic routing tree protocol was proposed. Dissemination of multiple queries with optimization is briefly discussed as the future work. TinyDB also discusses policies that have to be made when the data cannot be delivered due to time constraints and resource constraints.

In order to process the query more easily, the query-based middleware usually has certain network structures such as cluster or tree. For example, SINA is a query-based middleware for WSN. It assumes that the sensor nodes are formed into clusters and the underlying communication is attribute-based. On the highest level, it provides a SQLT for the users. SQLT is basically a SQL-like language and supports task scripting when the sensor meets the condition of the query. SINA is only evaluated in simulation and it doesn't seem to have any real implementation.

2.2.3 Virtual Machine-based Middleware

Virtual machine-based middleware usually provides an instruction set for the applications. As illustrated in Figure 2.5, the instructions serve as the interface for the high level applications while the actual services are provided by the underlying modules. Depending on their complexity, some modules may have more than one instructions linked to them.

Some of the existing works use virtual machine models that have already been developed such as JVM. For example, MagnetOS [7] provides the image of JVM 1.3. Two algorithms have been proposed to dynamically place the

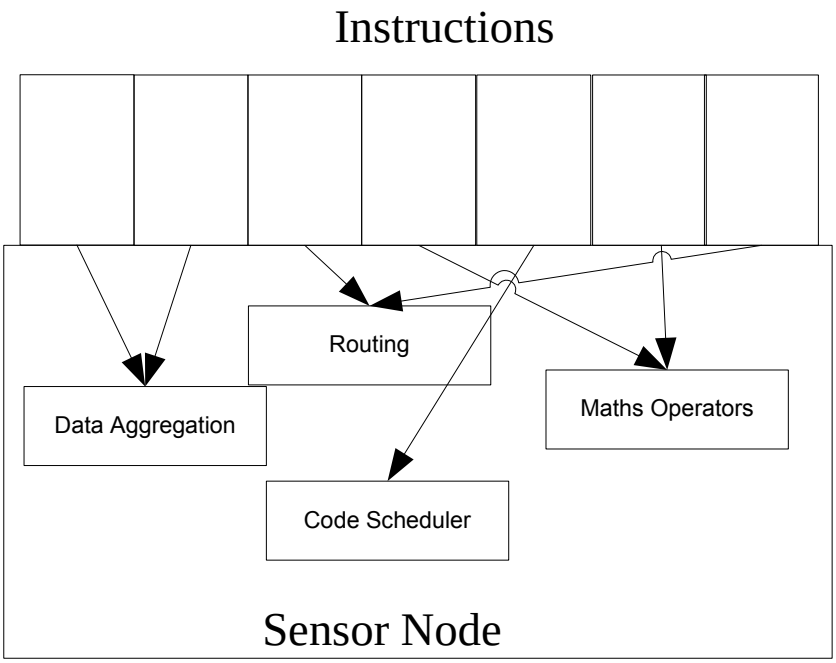


Figure 2.5: Virtual machine-based middleware for WSN

application modules on sensor nodes in order to save energy. MagnetOS seems to have been implemented but not on Berkeley sensor motes.

However, virtual machines such as JVM may be too heavy weighted to be executed on sensor nodes. Therefore, some work chose to develop new VM models. For example, Maté [64] is a VM based approach for WSN middleware. Three contexts are defined and they correspond to three events: clock timers, message reception and message send request. Each context has a stack structure to allow concise code size. Based on such kind of stack structure, coding in Maté is also stack-based. Maté has been implemented in TinyOS.

Other middleware may require supporting components. For example, Smart Messages (SM) [52] mainly contains three components. A VM is used

to provide hardware abstraction for SM execution. A tag space offers a name-based memory and consists of (name, data) pairs. A code cache is used to cache some of the codes for efficiency. With the help of these components, SM uses content-based routing and migration based on the tag spaces. Two routing algorithms have been implemented. One is based on Ad hoc On-Demand Distance Vector (AODV) [93], the other one is based on Greedy Perimeter Stateless Routing (GPSR) [53]. SM discusses security issues and provides the Admission Manager module to achieve the security goal. SM has been implemented on Pocket PC.

2.2.4 Event-based Middleware

Event-based middleware first allows the users to define events. Then the corresponding event handler will be specified and executed once certain type of event happens. Existing event-based middleware mostly deals with primitive events and many of them use SQL to define and subscribe to events.

A good example for event-based middleware using SQL is DSWare [66]. DSWare uses a SQL-like language to express describe events. The SQL commands 'insert' and 'delete' are used to register and cancel a specific event. However, with these simple commands, DSWare can only support primitive event detection. DSWare is implemented only in simulations.

In addition to events subscription and report, some other works make use of events to update codes. Impala [71] supports application module updates based on the reception of different kinds of events. In their paper, the authors focus on the architecture of the middleware Impala. No detail is provided

about the high-level API provided by Impala. There are three main modules in the middleware: Application Adapter, Application Updater and Event Filter. Impala has been implemented on Pocket PC with Linux operating system. The work is compared to Maté in terms of design and programming paradigm provided to the application layer. It didn't compare the actual performance.

2.2.5 Agent-based Middleware

An agent-based approach usually treats the codes running on individual sensor nodes as mobile agents. Codes can therefore, move between different sensor nodes. One of the motivations to use an agent-based approach is to allow multiple applications to be executed on the same network. One of such example is Agilla [27]. Agilla is based on Maté and uses stack architecture to reduce the length of the instructions. On each of the sensor node, a max number of 4 agents can be supported at the same time. Agents can be cloned or migrated depending on the application. Inter-agent coordination is achieved through a tuple space and an acquaintance provided by the Agilla middleware.

Different from previous middlewares, agent-based approaches usually do not have a de facto language or scripts and different work may adopt different approaches. For example, Agilla uses stack-based instructions. Another work SensorWare [11] uses a Tcl core as the scripting language provided to the upper layer. The program is basically composed of event handlers which perform tasks based on the events received. A mobile agent-based

approach is used in the sense that the script can be replicated and migrated in several sensor nodes. SensorWare seems to have a module, namely Admission Control module, for security. SensorWare has been implemented on Pocket PC. Its performance is briefly compared with SQLT (used in SINA) in terms of code size.

2.2.6 Middleware with Other Programming Abstractions

The middleware listed before use popular and well-understood programming abstractions. There are, however, other works that use programming abstractions which are more specific to certain applications or are simply not as popular as the previous ones. For instance, tuple space is a popular approach in traditional distributed systems but may not be as well-known as queries. TinyLIME [21] is a middleware that is based on tuple space. TinyLIME is built on top of LIME which uses a tuple space approach. The authors assume a network model where sensor nodes are sparsely scattered in a certain region. They communicate with a base station within its one-hop communication region. Base station and clients are connected with each other using multihop communication. The sensory data of a sensor node is represented as a mobile agent on the base station. TinyLIME has been implemented on both Berkeley motes and notebook with Java.

Some works develop programing abstractions which are specific to certain applications. For example, EnviroTrack [1] introduces a programming language designed specifically for object tracking in WSN. The EnviroTrack

compiler will first compile the program into C programs. The language has hid out the network layer to the application programmer and it provides an object-based naming mechanism. EnviroTrack has been implemented in TinyOS on Mica motes.

Some works may not emphasizes on programming abstractions but they do address some issues listed in Section 2.2.1, for example, data fusion through clustering. Yu et al. [111] specifically discuss the design issue of a cluster-based middleware. In this paper, the authors first discuss some general design principles for designing a WSN middleware. The discussed design principles include data-centric mechanisms, application knowledge, localized algorithms, lightweight and QoS. Based on these design principles, the authors argue that a cluster-based approach is more suitable to meet those design principles. The authors also briefly describe the architecture for such kind of middleware.

Apart from clustering, QoS is also an important topic in WSN. MiLAN [44] focuses on QoS and it introduces the concept of proactive network. In MiLAN, the authors argue that the previous works has focused on designing new network-level protocols without considering existing standards or how applications use the protocols. MiLAN extends the network stack and allows applications to specify the required variables and QoS requirements associated with them. A state-based graph is used to let the application describe variables. In the network layer, MiLAN uses a service discovery protocol. The implementation details haven't been discussed in the paper.

Cross-layer is also addressed by some middleware design. TinyCubus [82] features a cross-layer design. There are mainly three modules in Tiny-

Cubus. The first component, data management frameworks provides a set of standard data management and system components. The components are selected based on current system parameters, application parameters and optimization parameters. The second component, cross-layer framework, provides a generic interface to support parameterization of components using cross-layer interactions. It uses a specification language that allows for the description of the data types and information required and provided by each component. The third component, tiny configuration engine, is used to control the topology and the role of each sensor. Code distribution can use such kind of configuration information to distribute the code more effectively.

2.2.7 Summary of Existing WSN Middleware

Most of the existing works describe their abstractions in details. Existing approaches include query-based approaches [10, 99, 79], VM-based approaches [7, 64, 52], tuple space-base approaches [21] and mobile agent-based approaches [27, 11]. In addition to these well-known programming paradigms, some works provide some application-specific abstractions. For example, Enviro Track [1] provide a programming model which is suitable for object tracking applications.

In terms of network issues such as routing and data fusion, some of the works have briefly described the related issues. For example, SINA [99] uses a cluster-based approach to perform data aggregation and discuss the routing issues about how to deliver the data to a mobile subscriber. SM [52] uses content-based routing and migration based on the tag spaces. Two routing

algorithms have been implemented. One is based on AODV and the other is based on GPSR. Yu et al. [111] further argue that a cluster-based approach is more suitable. TinyLIME [21] briefly describe the communication scheduling issues for WSN where most of the sensors sleep during the most of the time. TinyDB [79] seems to have a most comprehensive discussion of the related network issues such as the construction and maintenance of semantic routing trees. The authors also discuss various kinds of policies for delivering the data.

In terms of implementation, some works [99, 66], seem to have been implemented only in simulation. For real world implementation, certain earlier works [71, 11, 52] make use of platforms such as pocket PC which are more powerful and expensive than those sensor nodes for large-scale deployment. With the increasing popularity for the Berkeley sensor motes [19] among academia, recent works [64, 21, 79, 1, 27] has been implemented on those less powerful sensor nodes. TinyCubus [82] has been implemented in TOSSIM [65] which is a simulator provided by TinyOS [31]. Codes created for TOSSIM can usually be deployed in real Berkeley sensor motes easily. Other works don't seem to describe the implementation details.

In terms of evaluation, most of the existing works don't have much formal mathematical models for analysis. Since middleware is usually targeted to a specific area of applications, it is also difficult to fairly compare the performance between middleware with completely different designs. As a result, most of the works don't compare with existing works. Instead, some works [99, 64, 11, 52, 27] evaluate the work by creating applications based on the middleware. Some works [71, 66, 111, 82, 21] evaluate the work by setting up

a testbed either in real world or in simulation and performing some experiments based on the predefined metrics. Other works [1, 79] are already very application-specific and simulations/experiments according to the predefined metrics can also be regarded as application simulation/experiments.

Recently, the security and QoS issues in WSN middleware have also attracted increasing attentions [44].

2.3 Macroprogramming for WSN

Apart from middleware approach, there are other works that design and implement different programming abstractions for WSN. Such works are different from middleware works because apart from programming abstractions, middleware works usually need to deal with other issues such as resource management, QoS and routing. As a result, most of the WSN middlewares are standalone programs by themselves with a lot of underlying mechanisms implemented such as tree or clustering. At the meantime, they also provide a thin high level programming abstraction (usually in terms of scripts) to the middleware users.

2.3.1 Overview

The works that we are going to discuss in this section focus primarily only on the programming abstraction. The programs written with such abstractions may be compiled into native binaries. Many of these works use the term 'macroprogramming' for such type of abstraction. More specifically, macroprogramming techniques usually introduce new and completely different view

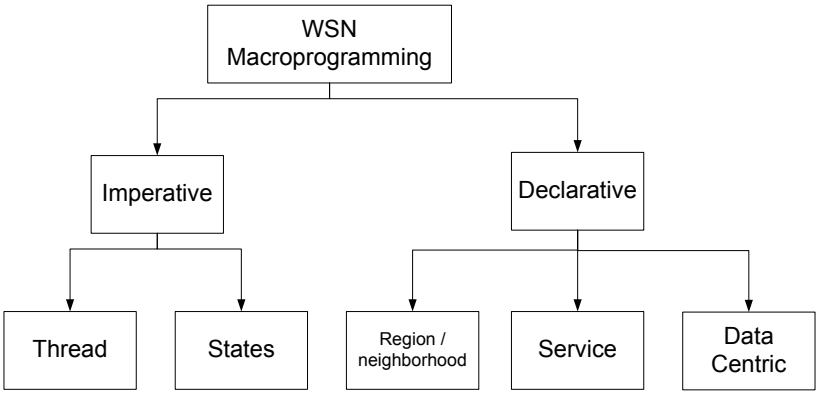


Figure 2.6: Categories of macroprogramming for WSN

on how to program sensor networks. They provide very high-level abstraction to the application programmers and let them to program the network as a whole. Some of the works on WSN middleware introduced in Section 2.2 can be regarded as macroprgramming even not explicitly mentioned. In addition to the works described in the previous section, we review some of the other works related to macroprogramming in WSN.

As shown in Figure 2.6, the existing works on macroprogramming may be first categorized as imperative and declarative. For imperative programming abstraction, the programmer will still have to specify the individual steps for performing the tasks. Under this category, we further have thread-based approach and state centric approaches. For declarative programming abstraction, we further divide them into three sub-categories: data centric, service and region / neighborhood based.

2.3.2 Abstraction for Imperative Programming

Due to resource constraints, implementing threads may not be so straightforward on sensor nodes since a lot of overhead may incur during context

Listing 2.1: A sample program written in RuleCaster

```
1  /* Ruleblock 1*/
2  SPACE(kitchen),TIME(1s){
3    STATE stoveOnHazard :- stoveOn(), notMonitored().
4    notMonitored() :- pressure(X), X<50.
5  }
6  /* Ruleblock 2*/
7  SPACE(door),STATE(kitchen:stoveOnHazard){
8    STATE hazard :- leaving(). }
9  /* Ruleblock 3*/
10 SPACE(door),STATE(door:hazard) {
11   ACTION alarm(10). }
```

switching. TinyThread [83] is a work that provides a library for threading. It proposes a multi-threading library for TinyOS, a popular OS used by sensor nodes. Basically, each thread has its own stack to store its local state. The programming abstractions include Blocking I/O and some synchronization primitives.

On the other side, state-based approaches borrow the idea from finite state machine. RuleCaster [9] is probably the first to describe such kind of abstraction. It consists of a high level state-based language, RCAL, a compiler that compile RCAL to codes executed by sensor nodes, and a middleware. An example of RCAL is shown in Listing 2.1

PIECES [69] took one step further by formally defining the state-centric programming model. It also uses the object tracking application as an example to show that the state-centric programming model is effective. Then, the idea of collaboration groups is proposed. Basically, a group contributes to a state update. Each group has a number of entities called agents. In PIECES, programmers think in terms of dividing the global state into a set

of pieces with one principal (a computational entity) maintaining each piece. Principals can communicate with each other by defining collaboration groups over principals. PIECES has been implemented only in Java and Matlab as simulations.

2.3.3 Service and Data Centric Abstraction

While imperative programming abstractions can probably give programmers enough expressiveness to define their application requirements, they still requires the programmers to write the detailed procedures. An alternative way is to provide declarative languages to programmers. Different from the imperative approach, in a declarative approach, programmers don't have to specify individual steps in their applications. Instead, they just need to declare what the application should do. A good example of a declarative language is SQL. To use SQL, the programmer does not have to understand the underlying details about how data are indexed and searched. They just need to specify what kind of data they want.

There are mainly three types of programming abstraction to be discussed in this section. The first one is the data centric approach. Apart from the query-based middleware introduced in the previous section, ATaG [91] is also a representative work in this category. Abstract task graph (ATaG) is a data-driven macroprogramming language. The language is composed of abstract task and abstract data item. Abstract channels are used to connect a task to a data item. In the paper, ATaG is expressed only in graphs and the authors briefly talk about the compilation process.

Listing 2.2: A sample program written in SNACK

```
1 service Service {
2   src :: MsgSrc;
3   src [send:MsgRcv] -> filter :: MsgFilter -> [send]
      Network;
4   in [send:MsgRcv] -> filter;
5 }
```

The service-based approach has also been studied by several works. This is partially because service is a hot research topic in WSN and there is a need for such a macroprogramming approach to support these research works. SNACK (Sensor Network Application Construction Kit) [34] consists of a configuration language, component and service library and a compiler. SNACK system leverages nesC [31] - its base components are written in nesC, and its compiler generates a nesC configuration and several nesC modules. A sample program written in SNACK is shown in Listing 2.2.

SNACK's syntax is briefly summarized as follows:

- $n :: T$ declares an instance named n of a component type T
- $n[i{:}\tau]$ refers to an output interface on component n with name i and interface type τ
- $[i{:}\ \tau]n$ refers to an input interface

The basic idea behind SNACK is dividing the programmers into three types. System programmers use nesC to develop reusable components. Service programmers combine those components into services that implement high-level semantics such as routing tree, periodic sensing, etc. Application programmers select a handful of services to run on a given network. A SNACK

Listing 2.3: Service description in SONG

```
1 service(breakService(Region),
2   needs(sensor(breakSensor,Region)),
3   creates(Pulse(X),detected(X,T,Region))
4 )
```

service, like a nesC configuration, is a collection of component declarations and connections that behaves like a component.

The actual definition of service, however, is different among different research works. As an effort to unify such a definition for services in WSN, SONG [70] formally describes the architecture and programming model of a semantic-service-oriented sensor information system platform. The authors think that by using ontology and semantic services, one may encapsulate sophisticated domain knowledge into computational components or services. Listing 2.3 shows such a service description.

For example, a SpeedEstimation service may take a sequence of pulses and produce the speed of the object. The implementation details of SONG haven't been discussed in the paper.

2.3.4 Neighborhood and Region Based Abstraction

The final category under declarative abstraction is neighborhood / region - based approach. Since the power of individual sensor nodes is extremely limited, it is very common for senor nodes to cooperate with their neighbors. Therefore, neighborhood-based approach can be useful for WSN applications. Some of the work provides a thin layer of library on top of TinyOS so that the programmers can make use of the neighborhood-based features in their

Listing 2.4: A sample program written in Kairos

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

familiar environment. Hood [106] is such an example. Developed by the same group that developed TinyOS, Hood proposes a neighborhood programming abstraction for sensor network. Sensors in the same neighborhood can share data among themselves. Hood provides a specification language that allows code to be generated in NesC programs.

While Hood provides a generic model for neighborhood-based programming, other works address issues within specific application settings. For example, Kairos [37] addresses the naming issue. It argues that in a distributed environment, sensor nodes may have conflicting names for resources. It ad-

dresses this issue by defining three simple programming abstractions. First, nodes are logically named using integer identifiers. Second, one-hop neighbors of a node can be obtained by calling get_neighbors(). Third, data at the remote node may be accessed by syntax like 'variable@node'. The code will be translated and linked to Kairos runtime and will be distributed by using code distribution software such as Deluge [46].

A sample program written in Kairos is demonstrated in Listing 2.4. In this short piece of codes, a shorted path tree is constructed by making use of the neighborhood abstractions provided by Kairos.

A similar concept to neighborhood-based approach is region-based approach. Since many WSN applications require location information, a region-based approach can be particularly important in these application areas. Abstract Region [105] provides a set of programming primitives for sensor networks. It is implemented in nesC as a library module. Listing 2.5 shows its APIs.

Listing 2.5: API provided by abstract region

```
1  /* Discover region */
2  result_t Region.formRegion(<region specific args>,
3  int timeout);
4  /* Wait for region discovery */
5  result_t Region.sync(int timeout);
6  /* Set local shared variable */
7  result_t SharedVar.put(sv_key_t key, sv_value_t val);
8  /* Get shared variable from give node */
9  result_t SharedVar.get(sv_key_t key, addr_t node,
10 sv_value_t *val, int timeout);
```

```
11 /* Wait for shared variable gets */
12 result_t SharedVar.sync(int timeout);
13 /* Reduce  value  to  result  with given op */
14 /*  yield  returns pct of nodes responding */
15 result_t Reduce.reduceToOne(op_t operator,
16 sv_key_t value, sv_key_t result,
17 float *yield, int timeout);
18 /* Reduce and set result in all nodes */
19 result_t Reduce.reduceToAll(op_t operator,
20 sv_key_t value, sv_key_t result,
21 float *yield, int timeout);
22 /* Wait for reductions to complete */
23 result_t Reduce.sync(int timeout);
```

A few applications such as object tracking and contour finding have been developed based on the API. A more formal definition of region is provided by Regiment [89]. On the highest level, a functional language called 'Regiment' is provided to the application programmer. A sample program written in Regiment is illustrated in Listing 2.6.

Listing 2.6: A sample program written in Regiment

```
1 let aboveThresh (p,x) = p > threshold
2   read node =
3     (read sensor PROXIMITY node,
4     get location node)
5 in centroid (afilter aboveThresh
6   (amap read world))
```

A compiler will then compile the 'Regiment' program into a program

that can be executed by a VM running on top of each sensor node. In their paper, the authors define several basic components, stream, space and event in Regiment. The notion of these components is based on Fran [22]. Based on these components, the authors propose another set of concepts such as area, region and anchor. A number of operators are defined for those components. In another paper, the same authors talk about their design of the Token Machine Language (TML) [88]. Basically TML is a virtual machine that can run on each individual sensor node. The basic components of TML are token store and handlers. Token store contains a number of tokens which can be regarded as events. The tokens can be sent and received through wireless channels. Handlers are software routines that execute when certain types of tokens are received. Some variables may be shared by multiple token handlers. An example of a handler which handles token 'Red' is defined in Listing 2.7.

Listing 2.7: A token handler for Regiment

```
1  shared int s;
2  token Red (int a, int b) {
3    stored int x;
4    if ( present(Green) )
5      x = 39;
6    else {
7      x += a + b;
8      timed schedule Red(500, s, a);
9    }
10 }
```

Up to now, the authors seem to focus on their design of macroprogramming architecture. A lot of detailed issues haven't been addressed especially how to build a compiler that compiles Regiment to TML.

2.3.5 Summary

Works on macroprogramming languages can be briefly categorized according to how they are used and deployed. Since the programming language, NesC [31] provided by TinyOS is already a powerful component-oriented language, some reusable modules with the NesC constructs to provide high-level abstractions. Some works such as [105, 83] use such kind of approach and they provide some kind of libraries to the application programmer to ease the development of applications. Some works such as [37, 34, 106] use a different approach. They provide a new language for the application programmer. Programs written in such kind of language will be compiled to NesC programs. Other works such as the works described in Section 2.2, [89, 9] move one step further. Instead of compiling the programs into NesC programs, they construct a runtime environment on the sensor nodes and compile the programs into intermediate programs that can be executed on such kind of runtime environment. Based on such observation, we define three different kinds of macroprogramming approaches:

1. Macroprogramming with library support: the approach that simply provides library APIs based on existing programming infrastructure such as NesC.

2. Macroprogramming with native code generation: the approach that

compile the programs into NesC programs.

3. Macroprogramming with runtime support: the approach that generates and distributes codes into the network with runtime support.

The first approach provides the lowest level of abstraction. With the help of code distribution tools such as Deluge [46], XNP [50] and Trickle [94], the second approach can achieve dynamic code distribution. For programs that require frequent updates, this approach may not be as efficient as the third approach because each time, the entire binary image of the program needs to be disseminated into the network. The third approach is the most flexible approach but the implementation of a runtime environment on sensor nodes with various resource constraints can sometimes be challenging.

Fortunately, we can reuse some existing works on WSN-based middleware to provide such runtime environment required by the last approach. Existing works such as Maté and Agilla provide low level yet compact instruction-like abstractions. We can provide higher level abstractions based on them. Moreover, these systems have been implemented in Berkeley motes and proven to work.

Existing works on macroprogramming in WSN may also be divided according to the programming primitives provided. Some works such as [37, 106, 105] provide neighbor-based primitives. Some works such as [34, 70] provide service-oriented abstractions. Some works such as [69, 9] provide state-base primitives.

Most of the works introduced in this section hasn't been implemented in the real hardware so there isn't much description of the performance evalua-

tion. Others use similar approaches as those described in WSN middleware such as by creating applications and testbeds.

2.4 Event-based Systems

As a first step towards a high-performance pub/sub middleware, we need to provide a high-level event language to application programmers. Such kind of language is primarily used to express complex events or event patterns. Unlike query-based middleware approaches which use standard SQL language, event definition languages should be more expressive for spacial and temporal relations among data. In this section, we take a look at the work that has already been made in this area. More specifically, we study the existing event-based systems from different aspects such as event definition, event evaluation and event operators / functions. These aspects are closely related to a successful event-based middleware.

2.4.1 Event Definition

Event definition may find its root in some earlier works on distributed systems since one of the issues that these systems deal with is distributed data processing. Some works such as Linda [14], use tuple space to represent simple events. Linda is a parallel programming language that makes use of tuple space. Four basic operations are defined in Linda: out(), in(), read() and eval(). In particular, eval is similar to event matching in pub / sub system. It matches actual (event) against formal (event expression). The use of actual and formal is also seen in WSN-based event systems such as low level

naming [42] and TinyLime [21].

Listing 2.8: Elvin's subscription language gramma

```
subscription:
   bool_expression
 | var_expression
bool_expression:
   "(" subscription ")"
 | "!" subscription
 | subscription "&&" subscription
 | subscription "||" subscription
 | "exists(" NAME ")"
 | type_expression equality_operator type_expression
type_expression:
   "string"
 | "int32"
 | "float"
 | "datatype(" NAME ")"
var_expression:
   numeric_expression
 | string_expression
numeric_expression:
   NAME numeric_operator INTEGER
 | NAME numeric_operator FLOAT
 | NAME numeric_operator NAME
string_expression:
   NAME equality_operator STRING
 | STRING equality_operator NAME
 | NAME equality_operator NAME
 | NAME "matches(" STRING ")"
```

```
28 equality_operator:
29     "=="
30   | "!="
31 numeric_operator:
32     "=="
33   | "!="
34   | "<"
35   | ">"
36   | "<="
37   | ">="
38 NAME: [a-zA-Z][a-zA-Z0-9_]*
```

However, the actual and formal described in Linda were more similar to primitive event matching and therefore, is not suitable for more complicated event systems. As a result, some later work defined events in a way which is more comprehensive than tuple space. For example, Meghdoot [38] defines an event as a conjunction of pairs, where each pair consists of an attribute and a value. A subscription is a collection of predicates each of which is a triple consisting of an attribute, a value and a relational operator.

A more formal specification of a subscription language is discussed in Elvin [97]. The language is both simple and straightforward. Its grammar is specified in Listing 2.8.

These works may be enough in some certain simple event systems where the event matching is only done on primitive events. On the other hand, systems that requires event patterns probably need more language features. Yeast [56] defines Event-action specifications. The users can define specifications that comprise event patterns. Each event pattern is a compound

pattern of primitive event descriptors. An event descriptor matches an event either transiently or permanently. For example, users can specify an event descriptor that is to be matched whenever the load on a particular computer host exceeds some threshold value, or an event descriptor that is to be matched after a specified time.

The Yeast specification language has the format of "event_pattern do action". In Yeast, events are categorized as time events, object events and compound events. Each kind of event has a set of descriptors. Time events have 'in' and 'within' which specifies the event is after of before a time. Object descriptors have the format of "obj_class obj_name obj_attr relational_test". In addition, two special relation tests 'changed' and 'unchanged' are defined. Compound event descriptors are combination of time and object events with 'then', 'and' and 'or'. An example of a Yeast script is shown in Listing 2.9.

Listing 2.9: Example of a yeast script

```
1 In 10 minutes and host research load > 5.0 do [something]
```

An more complicated example of Yeast with server is shown in Listing 2.10. Basically, the user can register with the server via 'regyeast'. After registration, a handful commands are provided to the user. 'addspec' is used to add an event specification. 'lsspec' is used to list the existing event specifications. 'suspspec' and 'fgspec' are used to suspend and resume a specific event specification.

Listing 2.10: Example of a yeast script with server

```
1 1% regyeast
2 you have been registered with yeast
```

```
3  2% addspec in 1 minute do echo 1 minute elapsed
4  3% lsspec
5    1 addspec in 1 minute do echo 1 minute elapsed
6  4% lsspec 1
7    1 addspec in 1 minute do echo 1 minute elapsed
8  Will attempt match at Sun Jan 1 11:39:28 1995
9  5% suspspec 1
10 8% lsspec
11   1 - addspec in 1 minute do echo 1 minute elapsed
12 7% fgspec 1
13 8% lsspec
14   1 addspec in 1 minute do echo 1 minute elapsed
15 9% sleep 60
16 10% 1sspec
```

To demonstrate the interactive usage of yeast, on Line 1 - Line 3 of Listing 2.11, one developer first defines an event that is used to monitor the debugging status of a source code file. Once the file has been debugged, on Line 4, the other developer announces the new status of the file and the corresponding developers will be notified.

Listing 2.11: Example of interactive yeast

```
1 defattr file debugged boolean
2 addspec file project.c debugged == true
3   do notify project.c debugged
4 announce file project.c debugged = true
```

Compared with previous works, a distinct feature of yeast lies in its capabilities in expressing temporal relations among events. Such feature is

important in a pub / sub system supporting composite events.

There are other works in the area of active databases that use event-condition-action (ECA) rules for event definition. In READY [35], the event specification is defined as: "event var : event type | expression". A few event operators are defined in the paper such as '&&' (and), '||' (or), ';' (then), and 'butnot'. Another form of compound matching is to specify a sequence of events which all match the same "element pattern" (a matching expression used for each element of the sequence) using: "event var[j::k] : event type | expression". The event var[j..k] indicates that the event variable will be bound to a sequence of events, with j and k being the minimum and maximum number of events in the sequence, respectively.

In SAMOS [30], rule definition is used to specify ECA-rules. The basic syntax is shown in Listing 2.12.

Listing 2.12: Syntax used in SAMOS

```
1 DEFINE EVENT event_name event_clause
2 DEFINE RULE rule_name
3 ON event_name IF condition_clause DO action_clause
```

The categories of events are described in the paper, primitive events and composite events which are described by event algebra. For composite events, six operators are defined. The basic ones are conjunction, disjunction and sequence. Other three operators include '*' which means the event will be only signaled once, history event which means the event will be signaled every time during a specific period of time and negative event which means the event will be signaled if it hasn't been detected during a specific period. Event

parameters are defined so that the origin of the event can be specified. For example: "(E1;E2):same object" denotes the sequence of events happening on the same object.

To make it easier for the user to understand the event definition, some works use the syntax such as query which is more popular. An example of such is Query for Events [12]. Listing 2.13 shows an example for of Query for Events.

Listing 2.13: An example of query for event

```
order[
  orderId{4711},
  customer{"John"},
  buy[
    stock{"IBM"},
    limit{3.14},
    volume{4000}]
]
```

A few operators such as conjunction and disjunction are defined and a formal definition of the query language is shown in Listing 2.14.

Listing 2.14: Formal language specification of query for events

```
DETECT bigbuy{
  tradeId{varI},
  customer{varC},
  stock{varS}}
ON buy{{
  tradeId{varI},
  customer{varC},
```

```
 8    stock{varS},
 9    price{varP},
10    volume{varV}
11  }} where { var P * var V >= 10000 }
12  END
```

With the increasing popularity for functional programming, some works such as λ-calculus [18] use functional approach. Functions in λ-calculus are anonymous and are written in the form "λx.E". λ indicates this is a function, x is the parameter and E is the body of the function.

These works from ADB may not directly address the issue of how to express the temporal and spatial relations among events. They do, however, provide some generic models such as ECA rules, event queries and λ-calculus that may be used to express more complicated relations among events. Such generic model can also be used in a pub / sub system for specifying composite events.

2.4.2 Event Evaluation

Once the event has been defined, it is also important to evaluate the event based on the event definition. In this section, we look at some of the works related to event evaluation. To make the evaluation simpler, some work may create specific data structures for evaluation. Event processing service (EPS) [85] provides an algorithm to evaluate composite events. Some of the basic operators are provided by EPS. The events are represented as trees. The leave nodes are primitive events and intermediate nodes are composite events or subexpressions provided by the application. The proposed algorithm allows

some common subexpressions to be shared by multiple composite events. A similar idea is proposed by Rebeca [86] that discusses a content-based routing algorithm. The algorithm is based on finding the relationships between filters and the paper discusses several propositions that can be used to find the covering and overlapping of conjunction filters.

Other than creating certain data structure for evaluation, some work try to group the event filters in a hierarchical way. SIENA [15] is such an example. In SIENA, the authors don't attempt to provide a complete pattern language. Their goal is to study pattern operators that can be exploited to optimize the selection of notifications within the event notification service. In the paper, they define various covering relations between filters, notifications and advertisements.

The work discussed previously are primarily intended for existing distributed systems and they may not consider the WSN-specific issues such as energy saving and deployment. As a result, they may not be suitable for WSN.

A popular approach for event evaluation in the field of active databases is the rule-based approach. As one of the most frequently referenced work, Generalized event monitoring language (GEM) [81] is used to define rules for monitoring complicated events. It allows abstract events to be specified in terms of combinations of lower-level events. GEM uses a declarative rule-based syntax in which various temporal constraints can be specified for event composition. Each GEM script is constructed from some basic GEM primitives. A few of them is defined in Listing 2.15

Listing 2.15: Primitives in GEM

```
1 primitive-event-expr :=
2   event-id | * | every <time-period-expr> | [at] <time-
      point-expr>
3 guarded-composite-event-expr :=
4   composite-event-expr [when guard]
```

Then, these primitives are connected using operators:

- e_1 & e_2: occurs when both e_1 and e_2 occur irrespective of their order
- e + time-period: occurs a specified period of time after the occurrence of event e
- $\{e_1 ; e_2\}$! e_3: occurs when e_1 occurs followed by e_2 with no interleaving e_3
- $e_1 \mid e_2$: occurs when e_1 or e_2 occurs
- $e_1 ; e_2$: occurs when e_1 occurs before e_2

Based on the operators, GEM scripts including event declarations, rule definitions and control commands can be defined with the syntax specified in Listing 2.16

Listing 2.16: GEM syntax

```
1 event <event-id>
2   [(<formal-attribute-declarations>)]
3 rule <rule-id> [<detection-window>]
4   { <event-expression> ==> <action-sequence> }
5 notify <event-id> [(<attribute-value-list>)]
```

```
6 forward (<event-id>|<event-variable>)
7 <event-id> [(<attribute-value-list>)]
8 enable [<rule-id>]
9 disable [<rule-id>]
```

As a practical example of using GEM, Listing 2.17 shows a GEM script that monitors gauge changes.

Listing 2.17: GEM example

```
1  event gauge_changed(double val)
2  event upper_exceeded
3  event lower_exceeded
4  rule gauge_rule1 { x:gauge_changed
5    when x.val >= 10 ==>
6      notify upper_exceeded;
7      enable gauge_rule2; disable }
8  rule gauge_rule2 { x:gauge_changed
9    when x.val <= 5 ==>
10     notify lower_exceeded;
11     enable gauge_rule1; disable }
12 enable gauge_rule2
```

Even though GEM is also for distributed systems, it's rule-based framework may also be applied to WSN since the detection of an event by one sensor node may trigger another sensor node to start event detection. In fact, its rule-based framework inspired the design of our event detection framework which will be discussed in Section 3.2.

Finally, some works use application specific event evaluation methods. EVE [32] is an event-driven middleware for workflow execution. Workflow

type specifications are mapped to B/SM. Each step in the workflow corresponds to the execution of a service. ECA rules define when and how services are executed. From the perspective of B/SM, a workflow consists of interacting, reactive components called brokers and broker behavior is defined by ECA-rules describing their reactions to events. The authors define some primitive events such as broker interaction events and time events. These primitive events can be constructed using some operators defined by the authors, such as sequence, exclusive disjunction, conjunction, repetition, negation and concurrency.

2.4.3 Event Operator and Function

Having discussed event definition and event evaluation, we now review some works on event operators and functions. We review the works in this category in order to summarize the necessary event operators / functions expected from a pub / sub system. While some of the works in the previous sections have already mentioned some event operators, in this section, we summarize the event operators from these works with a deeper discussion.

When active database (ADB) started to become a hot topic, it was soon realized that composite events are needed to specify events in such a system. SNOOP [16] is such a system for ADB. the event operators defined in SNOOP are shown as:

- OR ($\bigtriangledown$): disjunction of two events E_1 and E_2, denoted $E_1 \bigtriangledown E_2$, occurs when E_1 occurs or E_2 occurs.

- AND ($\bigtriangleup$): conjunction of two events E_1 and E2, denoted $E_1 \bigtriangleup E_2$,

occurs when both E_1 and E_2 occur, irrespective of their order of occurrence.

- SEQ (;): sequence of two events E_1 and E2, denoted $E_1; E_2$, occurs when E_2 occurs provided E_1 has already occurred. This implies that the time of occurrence of E_1 is guaranteed to be less than the time of occurrence of E_2.

- Aperiodic Operators (A, $A*$): The Aperiodic operator A allows one to express the occurrences of an aperiodic event within a closed time interval. There are two versions of this event specification. the non-cumulative aperiodic event is expressed as $A(E_1, E_2, E_3)$, where E_1, E_2 and E_3 are arbitrary events. The event A is signaled each time E_2 occurs within the time interval started by E_1 and ended by E_3.

- Periodic Event Operators (P, $P*$): A periodic event is a temporal event that occurs periodically. A periodic event is denoted as $P(E_1, TI[: parameters], E3)$ where E_1 and E_3 are events and $TI[: parameters]$ is a time interval specification with optional parameter list. P occurs for every TI interval, starting after E_1 and ceasing after E_3. Parameters specified are collected each time P occurs. If not specified, the occurrence time of P is collected by default.

- NOT ($\neg$): the NOT operator, denoted $\neg(E2)[E_1, E_3]$, detects the non-occurrence of the event E_2 in the closed interval formed by E_1 and E_3

Apart from event operators, some works use event functions which is

conceptually similar to operators but syntactically different. Java Event CorrelaTOR (JECTOR [68]) tries to use composite event specification approach that can express complex timing constraints among correlated event instances. Event specification in JECTOR is shown in Listing 2.18

Listing 2.18: JECTOR syntax

```
attributes ( [NAME, TYPE], . . . , [NAME, TYPE] )
which occurs
whenever timing condition
  TC is [satisfied I violated]
if condition
  C is true
then
  ASSIGN VALUES TO CE s ATTRIBUTES;
```

An example of a composite event using JECTOR is shown in Listing 2.19

Listing 2.19: JECTOR example

```
define composite event LinkADownAlert with
      attributes (["Occurrence Time" : time] ,
      ["Link Down Time" : time]) which occurs
whenever timing condition
  @(LinkADown, i) + 2 minutes <= @r(LinkAUp, @(LinkADown,
     i), I)
  and @(LinkADown, i) + 2 minutes <= @r(LinkADownAlert,
  @(LinkADown, i), I) and @(LinkADown, i) - 3 minutes >=
  @r (LinkADownAlert , @ (LinkADown, i) , 0)
  is satisfied
if condition true is true
then {
```

```
12    "Link Down Time" := @(LinkADown, i) ;
13  }
```

A few works [108] summarizes the existng event operators. They compared a lot of existing works on composite event description and proposed some unified event operators. The work first summarized the existing event operators as in Table 2.1.

	Operators						
	Conjunction	Disjunction	Sequence	Concurrent	Negation	Iteration	Selection
ECCO	$A + B$	$A \mid B$	$A; B$	$A \mid\mid B$	$\neg A$	$A*$	A^N
Opera	$A \mid\mid B$	$A \mid B$	$A; B$	-	$\neg A$	$A*$	-
CEA	$A \& B$	$A \mid B$	$A; B$	-	$\neg A$	-	-
Schwiderski	A, B	$A \mid B$	$A; B$	$A \mid\mid B$	NOT A	$A*$	-
A-mediAS	$A \& B$	$A \mid\mid B$	$A; B$	-	$\neg A$	-	$A^{[i]}$
Ready	$A \&\& B$	$A \mid\mid B$	$A; B$	-	not A	-	-
Eve	$CON(A, B)$	$DEX(A, B)$	$SEQ(A, B)$	$CCR(A, B)$	$NEG(A, B)$	$REP(A, n)$	-
GEM	$A \& B$	$A \mid\mid B$	$A; B$	-	$!A$	-	-
Snoop	A, B	$A \vee B$	$A; B$	-	-	$A*$	-
Rebeca	$A \wedge B$	$A \vee B$	-	-	$\neg A$	$A*$	-
SAMOS	A, B	$A \mid B$	$A; B$	-	NOT A	TIMES(n, A)	$A * / last(A)$

Table 2.1: Summary of existing event operators

Based on the summary for the existing operators, a set of unified operators are proposed as follows:

- Conjunction $A+B$: Event A and B occur in any order. $(A+B)T$ with a temporal parameter T indicating the maximal length of the interval between the occurrences of A and B. Note that $(A+B)\infty$ or $(A+B)$ refers without restrictions.
- Disjunction $A|B$: Event A or B occurs.
- Concatenation AB: Event A occurs before event B where timestamp constraints are A meets B, A overlaps B, A finishes B, A includes B, and A starts B.
- Sequence $A;B$: Event A occurs before B where time stamp constraints are A before B, and A meets B. $(A;B)0$ is a special case belonging to A meets B.
  - E.g. $(A;NULL;B)$: denotes there is no occurrence of any event between event A and B.
  - E.g. $(A;B)T$: means that an interval T between event A and B.
  - E.g. $(A;B0)$: denotes that there is event A and event B occur contiguously.
- Concurrency $A||B$: Event A and B occur in parallel.
- Iteration $A*$: Any number of event A occurrences.
- Negation $\neg AT$: No event A occurs for an interval T.

  - E.g. $(A\neg B)$: denotes no B occurs during A's occurrence.
  - E.g. $(A\neg B)T$: denotes no B occurs after starting A's occurrence within an interval T.
  - E.g. $(A;B)\neg C$: denotes that event A is followed by B and there is no C in the duration of $(A;B)$.

- Selection AN: The selection AN defines the occurrence defined by N.
  - E.g. $AAVGT$: denotes taking the average during an interval T.
  - E.g. $ALASTT$: denotes taking the most recent instance during an interval T.

- Spatial Restriction AS: Event A occurs if it is a spatial restriction defined in S, that can be defined as a specific location or a group identifier etc.
  - E.g. $ACB03FD$: The area code CB03FD identifies the zone around Computer Laboratory in Cambridge. Event A is valid only when spatial condition is satisfied.

- Temporal Restriction AT: Event A occurs within T.
  - E.g. $(A;B)T$ or $(A;BT)$: B occurs within an interval T after A.
  - E.g. BT: B is valid for an interval T.

They also discuss the issues about how to detect events based on FSA.

2.4.4 Summary

Most existing works on distributed pub/sub systems such as [15, 38, 97] only provide primitive events filtering. Each event is usually regarded as a set of attribute-value pairs. Each attribute usually has an identifier and a data type.

In the field of active database (ADB), there are also quite a few works on event description languages [68, 16, 35, 30, 86]. ADB extends the traditional database by incorporate the ECA rules. ECA means 'on Event; under Condition; take Action'. ECA is used in ADB for describing complex user preferences. An example of ECA is that customer may define their requirements as 'When the stock price of IBM has dropped to 50, buy in 100 shares". The most difficult part of ECA is the event description so quite a few works in ADB propose different solutions to this problem. Event description languages based on ADB usually focus on expressing complex temporal relationships between events. A lot of temporal operators have been defined and they may be borrowed when designing our own event description language. However, these languages normally don't provide any mechanisms for describing spatial relationships between events since spatial relationships don't make much sense in a database system. Therefore, we probably need to use a more general approach for our event description language.

In terms of implementation and evaluation, some of the works in ADB is probably not of our interest since they belong to another field. For example, one of the works in ADB implemented the language based on the log file of the database. This is probably not applicable in WSN. For other works

some of them did a real compiler implementation and briefly discuss the implementation issues.

Some works such as [81, 108] try to provide a general model for event description. [56] is also an interesting piece of work in the sense that it proposes an interactive specification language for event description.

In summary, events may be divided into primitive events and composite events. Each primitive event is composed of a set of attribute-value pairs. Each composite event is consisted of a set of primitive events which are combined with a set of operators. [16, 108] have detailed descriptions of these operators. Almost all existing works on event description language try to invent its own syntax and notion.

2.5 Data Aggregation in WSN

Data gathering technique is important to the design and implementation of PSWare because event detection is also a special type of data gathering. In order to reduce the traffic, data aggregation is widely used to process some data in-network. The data aggregation problem in WSN is well-studied. There are a lot of existing works trying to solve this problem from different perspectives.

The existing works on data aggregation may be categorized according to the network layer they operate on. In particular, since there are many works which use data aggregation on network layer, we further divide the data aggregation approach on network layer into tree-based and cluster-based approaches for a clearer presentation.

2.5.1 MAC Layer Data Aggregation

In MAC layer approach, data are aggregated without prior knowledge of application or network. The benefit of this approach is that the approach can be quite generic since it is independent of other layers. In its simplest form, the approach just opportunistically merge multiple packets into one to save some overhead. A representative example of this is adaptive application-independent data aggregation (AIDA) [40]. AIDA is an application independent aggregation protocol which lies between MAC layer and network layer. The basic idea is to combine multiple network packets into a single MAC layer packet to transmit in order to save the MAC layer control messages such as RTS and CTS. Aggregation occurs based on the feedback of the traffic information. Actually, such kind of energy saving is very limited since combining multiple small packets into a big packet will introduce extra cost if the transmission fails and retransmission is necessary.

A similar work in MAC layer approach is Data-Aware Anycast (DDA) [25]. The work introduces two aggregation protocols at different layers. DDA is the MAC layer aggregation approach. An 'aggregation ID' which is a timestamp is included in the RTS message. The node which can aggregate the packet replies with CTS. Randomized Waiting (RW) is another approach at the application layer. It allows the sensor nodes which are closer to sink to wait for a longer period of time in order to aggregate packets.

However, because the MAC layer approach does not consider information from other layers, there is a limit on which the data can be aggregated. In view of this, Tree on DAG (ToD) [24] took one step further. ToD is based

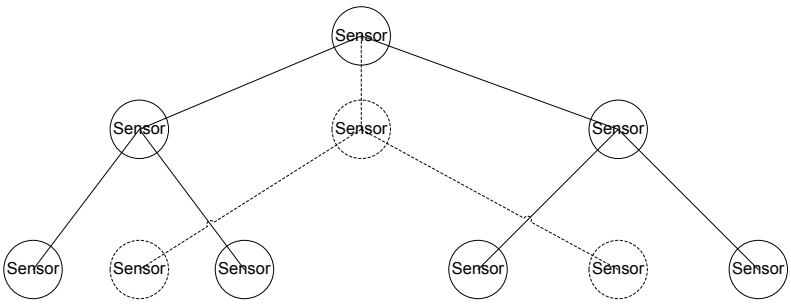


Figure 2.7: Basic idea of ToD

on DAA. The authors further propose an idea which combines structure-free and structured aggregation techniques. The basic idea is illustrated in Figure 2.7. It divides the network into cells. Within each cell, structure-free aggregation is used. Inter-cell aggregation is done by a structure called ToD. ToD is basically a combination of two trees. The following figure shows an example of ToD in one-dimension. In case the event occurs across multiple cells, the two trees ensure that the data can be aggregated at either one of them.

2.5.2 Cluster-based Data Aggregation

Clustering is a popular topic in many WSN-based applications. It may be considered as a network layer technique that groups sensor nodes to perform certain tasks. Clustering is a natural step to data aggregation in the sense that clusters can allow sensor nodes to send their data to the cluster head for distributed processing. Clustering for data aggregation has been studied for a relatively long period of time. Low-energy adaptive clustering hierarchy (LEACH) [43] is probably one of the earliest work. It is an application-specific power-efficient routing protocol and is suitable for remote monitoring

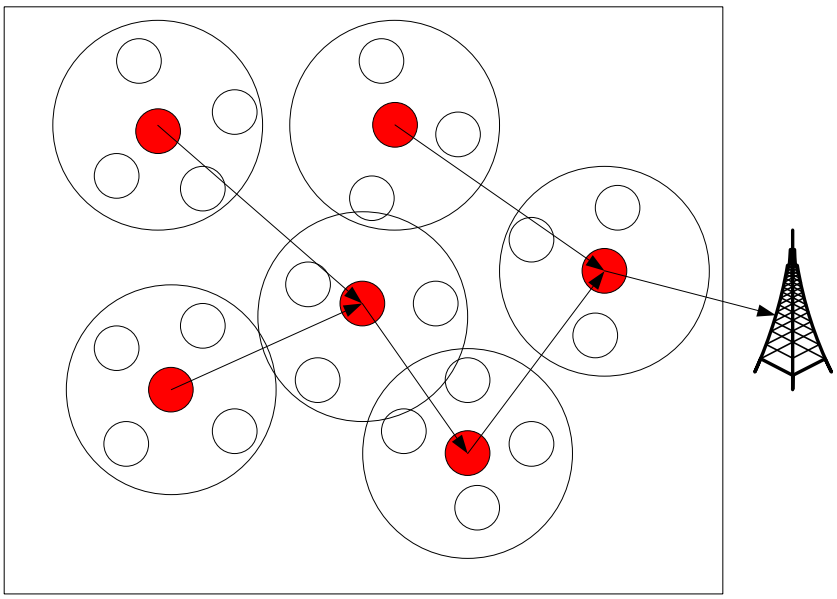


Figure 2.8: Clustering in iHeed

applications. The protocol is divided into rounds. In each round, the cluster heads are altered. However, the protocol is based on the assumption that all sensor nodes could adjust their power in order to communicate with the sink.

Power-Efficient GAthering in Sensor Information Systems (PEGASIS) [67] was introduced as an improvement of Leach. Similarly, it is still based on the assumption that each sensor node has power control and the ability to transmit data to any other node in the network. The basic idea of PEGASIS is that further energy saving can be achieved if data is aggregated towards the sink. The problem can be essentially formulated as a traveling sales person problem. A communication chain is formed for only once first. Each sensor node must have the global knowledge of the network. A TDMA scheme is used to allocate time slot for each sensor node to transmit data.

As sensor nodes have become smaller, it has come clear that for many

WSN-based applications, it may not be easy or cost-effective to adjust sensor nodes' transmission power once deployed. Therefore, more practical implementation was needed. Integrated Hybrid, Energy-Efficient, Distributed (iHeed) [109] is a real implementation of a cluster protocol. The protocol is integrated into the existing multi-hop routing protocol of TinyOS. The protocol periodically uses a probabilistic approach to elect cluster heads with high residual energy. By using such an approach, the clusters heads in iHEED are well distributed as shown in Figure 2.8.

Apart from real implementation, some works address QoS requirements when clustering [113]. Depending on the time delay, three cases may occur:

- If the delay constraint can be satisfied, the sensor node defers the report for a fixed time interval with certain probability. When delay occurs, the sensor node will receive and aggregate any additional reports.

- If the delay constraint can be satisfied only if the report is not deferred, the sensor node simply tries to forward this report to the next hop.

- If the delay constraint cannot be satisfied in any case, the sensor node will discard the report, to avoid further wasting of any additional resources.

A formal study of the distributed clustering problem is presented in Energy-Efficient Protocol for Aggregator Selection (EPAS) [17]. It studies the problem on how to select aggregators evenly in the network in a distributed fashion to achieve best energy saving. The authors start their algorithm from one-level aggregation first. Then, the work is further extended

to Hierarchical EPAS (hEPAS) to provide a multiple-level aggregation solution.

Because of the vast amount of works done in cluster-based data aggregation, some work summarizes the existing techniques [110]. In terms of how to elect cluster heads, there are four metrics: node ID, larger degrees, higher weights and node redundancy. The authors suggest that metrics based on node ID or degree may not be energy efficient because certain nodes in the network may exhaust their energy faster than others. In terms of execution of the clustering algorithms, it can be either iterative or probabilistic. In iterative clustering algorithms, a node waits for a specific event to occur to decide its role. Iterative algorithms usually result in slow convergence speed. In probabilistic clustering algorithms, nodes independently decide their roles. There are still a lot of open issues such as connectivity, MAC, rotating CH, node duty cycle, optimal cluster size and node synchronization.

2.5.3 Tree-based Data Aggregation

Tree-based data aggregation is another aggregation technique in the network layer. Different from clustering, the degrees of the nodes in a tree are usually more evenly distributed than those in clusters. Many problems in this area can be formulated as classic graph theory problems or optimization problems. Some of the early work in this area took an intuitive approach [55]. They introduce and analyze a few data-centric aggregation schemes. These schemes include 'center at nearest source (CNS)', in which all sources send data to the nearest source to the sink, 'shortest paths tree (SPT)', in which the data are

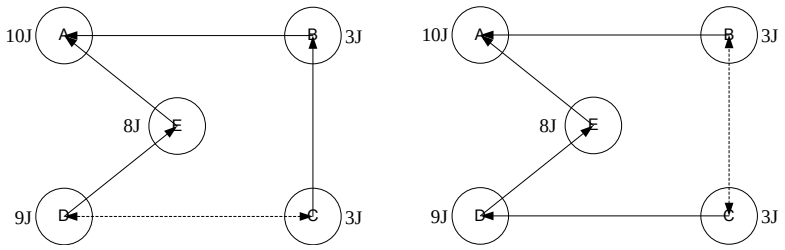


Figure 2.9: Optimized tree construction based on residual energy

sent along the shortest paths and aggregated on the common edges, 'greedy incremental tree (GIT)', in which sources to the sink are added to the tree one by one based on their distances to the sink.

Some similar ideas were discussed in [92] where the authors first divide the existing aggregation works into two general categories. Routing-driven compression (RDC) routes the data with the shortest path and opportunistically perform aggregation. Compression-driven routing (CDR) tries to optimize the routing tree to compress more data. The methodology based on joint-entropy. The author uses such a mathematic framework to evaluate these routing schemes under different parameters. Based on their results, the authors then propose a sub-optimal scheme which forms several clusters in the network. The sensors within each cluster uses shortest path to send data to the cluster head. The cluster head then aggregate data and send to sink without any further aggregation. [102] is similar to [92]. The authors take one step further and make a different and more reasonable assumption that the sensors be randomly deployed in the area.

Optimization techniques are widely used in tree-based approach. Many works in this area formulate the problem into mathematical programming problem. For example, Xue et al. [107] considers the residual energy and the

transmission cost between individual nodes. Then the problem is formulated as a linear programming problem (LP) and is extended to multiple sink nodes. Figure 2.9 illustrates the tree construction process.

Similarly, [63] also considers residual energy but formulate the problem into another graph theory problem. It lets the nodes with higher energy to be the aggregation parents. The paper argues that residual energy must be considered in order to maximize network lifetime. An example is shown in the paper as the following figure. The shortest path from C to A is through B but since B has little energy left, this link will end prematurely if C connects to B. A better approach is to let C connect to D which has higher residual energy. Two approaches are proposed, one centralized approach and the other distributed approach. In the centralized approach, all sensor nodes are first sorted according to their energy levels. The problem is then naturally transformed to a graph theory problem in order to find optimal tree based on the existing energy level and topology. The distributed approach is similar to Reverse-Path Forwarding (RPF). There are two steps in this approach.

Apart from residual energy, other parameters may also be considered in formulating the problem. Minimum Fusion Steiner Tree (MFST) [75] uses Steiner tree to model the problem. The work considers both transmission cost and fusion cost which is equivalent to processing cost. The work was later extended to cope with network dynamics.

The tree construction problem can be more complicated if mobility is taken into consideration. Dynamic Convoy Tree-Based Collaboration (DCTC) [112] is a work that studies the tree construction problem for mobile WSN applications such as target tracking. The algorithm is used to collect local

data of the sensor nodes to a mobile sink. The protocol sets up a convoy tree with the root close to the mobile sink. There are two schemes proposed in the protocol, conservative and prediction-based. Conservative scheme add the sensor nodes to the convoy tree based on their distances to the mobile sink while prediction-based scheme adds sensor nodes to the convoy tree based on a prediction algorithm. The tree will be reconfigured when the mobile sink moves further away from the original root.

2.5.4 Application-specific Data Aggregation

If the application level knowledge is taken into consideration, further energy efficiency may be achieved. However, each data aggregation technique introduced in this section can only be applied to a specific type of WSN applications.

The first type of data aggregation technique we are going to introduce is related to our previous sections since it is related to query-based middleware for WSN. Tiny Aggregation Service (TaG) [78] is a work which was later extended to TinyDB. The work tries to add aggregation service of WSN to the core service. The aggregation is expressed in SQL and is disseminated from the sink. Each sensor node chooses its sender of the query as the aggregation parent. During aggregation, parents will wait for the children's messages before sending their own.

Based on the similar concept, Temporal coherency-aware in-Network Aggregation (TiNA) [98] is another work based on Cougar [10]. It makes two contributions. First, Group-Aware Network Configuration (GaNC) is a rout-

Listing 2.20: An example of TiNA

```
1 SELECT {attributes, aggegates}
2 FROM sensors
3 WHERE conditions-A
4 GROUP BY {attributes}
5 HAVING conditions-B
6 EPOCH DURATION i | EVERY e
7 TOLERANCE tct
```

ing protocol that builds the routing tree according to the 'group by' clause of the SQL in order to save energy. It is compared with the 'First-Heard-From (FHF)' method. During parent-selecting, the node will also look at the (static) attributes used in the 'group-by' clause to select a parent with the same group. TiNA introduces a new type of SQL clause for WSN: TOLERANCE x%. When a new sensor's reading is within the difference of x%, it will not be transmitted. TiNA can be built on top of any existing SQL-based aggregation techniques. The language extenstion demonstrated in Listing 2.20. The standard SQL part is from Line 1 to 5 and the tolerance introduce by TiNA is at Line 7.

Another application specific work is VigilNet [41]. It was later extended to the EnviroTrack middleware [1] which was briefly introduced in the previous section. VigilNet had an in-depth discussion on how to implement a real data aggregation system for the tracking applications. There are four layers of aggregation in the proposed system. Level 1 mainly concerns how to sample and aggregate raw data. Level 2 concerns how to aggregate data from different sensors on a single node. Level 3 concerns how to aggregate data within groups. Basically, potential group leaders are selected based on

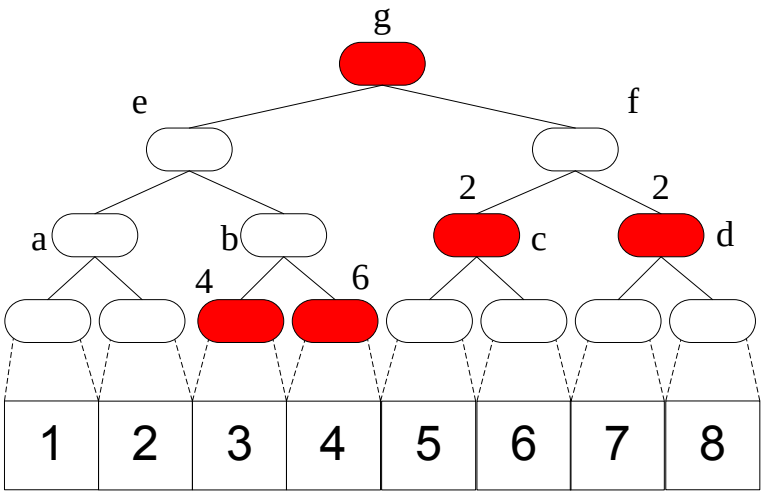


Figure 2.10: An example of q-digest

max coverage. Level 4 is just the backend aggregation where all the data has been transmitted to the sink.

Instead of using application knowledge to help the underlying network layer to aggregate data, some works directly use statistical methods to directly reduce the data volume. Such works aim at finding the optimal trade-off between data accuracy and energy efficiency. For example, in paper [51], the authors propose the algorithms which can use any multi-path routing protocols for aggregation in order to increase reliability. It tries to capture the distribution of all the data and aggregate the parameters only. The work is based on Expectation-Maximization (EM) algorithm, which is standard for finding maximum likelihood estimates of parameters in probabilistic models.

Q-digest [100] is a data structure which can be used to estimate some more complex aggregation functions such as median, histogram and so on. Basically, the data structure is a binary tree structure with every node represents the number of values for a specific range. The leaf nodes represent

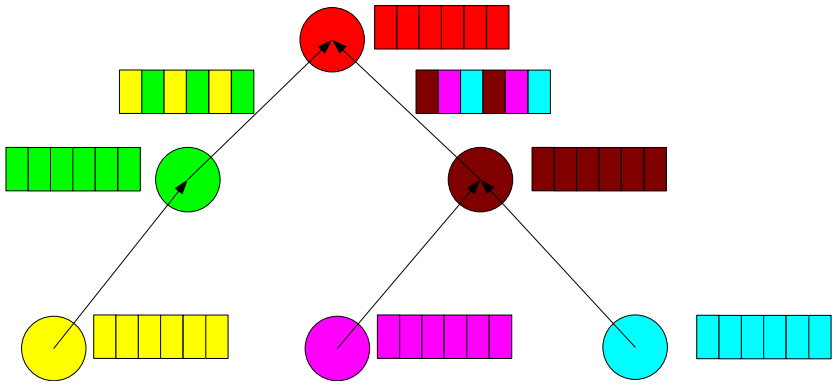


Figure 2.11: Example of deterministic weighted sample

the actual values. Instead of retaining all the leaf nodes, it tries to balance the trade-off between accuracy and space by introducing intermediate nodes. Each intermediate node represents a summary of its child nodes. In the case of q-digest, each intermediate node represents a range and the number of child nodes within that range. Only those intermediate nodes will be actually stored and processed. An example of Q-digest in shown in Figure 2.10. The trade-off discussed in q-digest is one of the primary concerns in WSN.

A quite similar idea is presented in [2] where tree-like structure is used to find the most representative data through deterministic weighted samples. The idea, however, is different from q-digest and can be considered as an improvement over q-digest. The algorithm tries to evenly distribute the sample data among all nodes in the sensor network instead of having highly concentrated sample data at the nodes near sink. As shown in the figure below, by assigning different weights to the sensor nodes, the final aggregation result will contain most representative data which is evenly sampled from all the nodes. Figure 2.11 shows an example where all nodes have the same weight. The algorithm not only tries to balance the trade-off between accuracy and

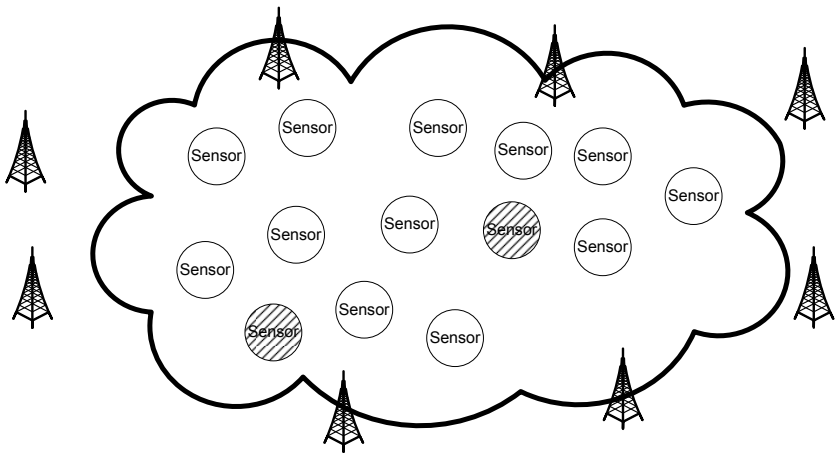


Figure 2.12: Network model of sparse data aggregation

space like q-digest, it also tries to evenly distribute the sample data among all nodes in the network. Such consideration is quite relevant to the scenarios in WSN if all sensor nodes are considered even and deployed randomly.

The last work we will introduce here is sparse data aggregation [29]. As the name suggested, the data aggregation techniques described is primarily for applications where the data are sparsely distributed among the network and the whole WSN is surrounded by high-speed network access. It is quite recent and addresses the problem of how to aggregate data from a set of source nodes which are sparsely distributed in the network. However, the work makes a very strong assumption that all the sensor nodes on the boundary can access an external high-speed network. This assumption is almost equivalent to assume that all the nodes on the boundary of the network are sink nodes. Figure 2.12 shows such a model described. The shaded nodes are the nodes with data to send. Some nodes are not in the tree but are involved during the tree construction phase. All the trees are rooted from the boundary of the network.

2.5.5 Summary of Existing Works

Some of the earlier works such as LEACH and PEGASIS [43, 67] make assumptions which are not suitable to current WSN. For example, both of the works make the assumption that any sensor node could adjust its power and communicate directly with the sink when needed. This may not be the case for the current mainstream of the research in WSN.

Some of the works are interesting but may not be so related to our event detection algorithms. For example, DCTC [112] gives very detailed mathematical model for the aggregation problem where the aggregator is mobile. Such kind of model has not been applied to our work yet.

There are a few works which may be quite related to our problem and can be leveraged when we design our own algorithms.

- MAC-layer aggregation: AIDA and DAA [40, 25] focus on simple data aggregation on the MAC-layer. These protocols are usually application independent. However, since the application knowledge is not present at such a low layer, these protocols can usually just save limited amount of energy. Energy saving of these protocols is usually done by simply combining multiple MAC packets into one so that control overhead such as RTS and CTS can be reduced.

- Energy-based clustering: different from MAC-layer approach, works such as LPT, iHEED and EPAS [63, 107, 109, 17, 75] take application knowledge into consideration. The main objective of these clustering algorithms is to save energy.

- QoS-aware clustering: different from energy-based approach, Zhu et

al. [113] discussed an aggregation algorithm with QoS as its primary objective.

- Database solutions: these approaches are more application-specific. TAG and TiNA [78, 98] belong to this category. These works support common aggregate operators in database systems such as average, minimum and maximum.

- Mathematic-based solutions: based on database solutions, mathematic-based solutions move one step further. In addition to database aggregate operators, these works also support more sophisticated aggregates such as median and histogram. Q-digest and some others [100, 2, 51] are representative works falling into this category.

- Double-ruling [29] may be the most relevant work to our problem. The work considers how to aggregate events occurred sparsely in different part of the network. Unfortunately, this work is based on a very strong assumption that all nodes on the boundary of the network be the sink nodes. Therefore, it is still unsuitable to model the event detection problem in our work.

In summary, none of the existing work can really solve our event detection problem efficiently. However, some of the works can still be leveraged and their results may be utilized in our work. Although data aggregation is a well-studied topic in WSN, we believe the performance of data aggregation can still be improved if we use a holistic approach. As shown in the later sections, by integrating application-level knowledge into the aggregation algorithm,

aggregation performance can be improved in most cases. We will discuss our solution in the next section.

Chapter 3

System Design

3.1 PSWare: Model and Architecture

In this section, we describe the application programming model for PSWare. Our model aims at solving the challenges which face event-based programming.

As shown in Figure 3.1, our model mainly has two different types of people that make use of two APIs of different levels:

- Application developers make use of the PSWare-EDL to define and subscribe events and use the event receiving module to receive the published events.

- Middleware developers make use of the event processing framework to implement effective and efficient event processing algorithms. The event processing framework may be further divided into two layers for event detection and event delivery.

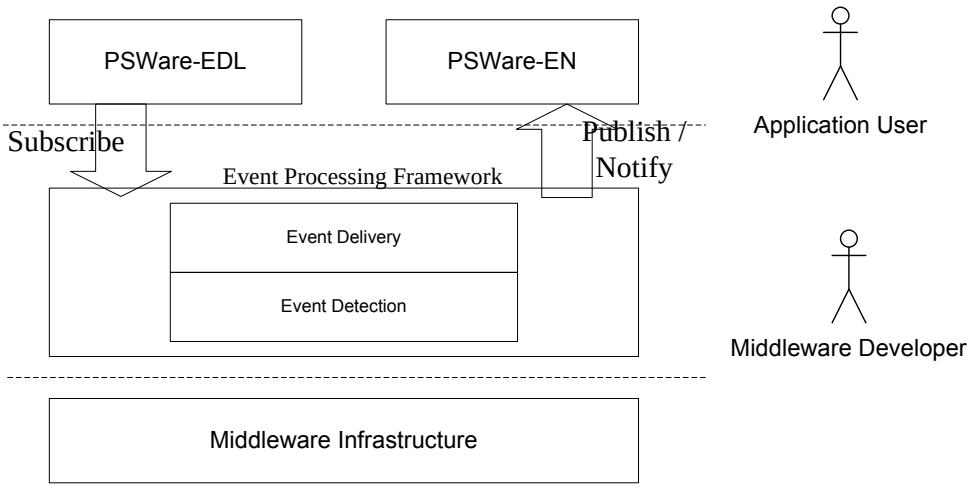


Figure 3.1: PSWare programming model

The first type is the application users. These users are responsible for defining and subscribing to high level events for different types of applications. They use a high level event definition language (PSWare-EDL) to translate the application requirements into events. They do not have to worry about the underlying event processing mechanisms.

On the lower level, we have another type of users called middleware developers. These users are responsible for implementing domain-specific event processing mechanisms. PSWare provides a couple of interfaces in TinyOS to make the implementation easier.

The benefit for such a model lies in its flexibility. There are usually many different types of applications for a specific application domain. For example, in ITS, we may have collision warning, traffic flow control and overspeed detection yet these applications can probably share a lot of common event processing mechanisms. Therefore, the middleware developer only needs to implement the event processing mechanism for once. Then, by defining dif-

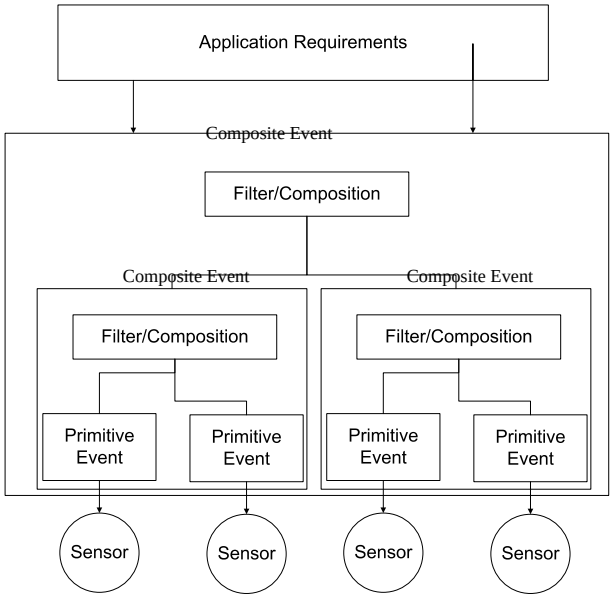


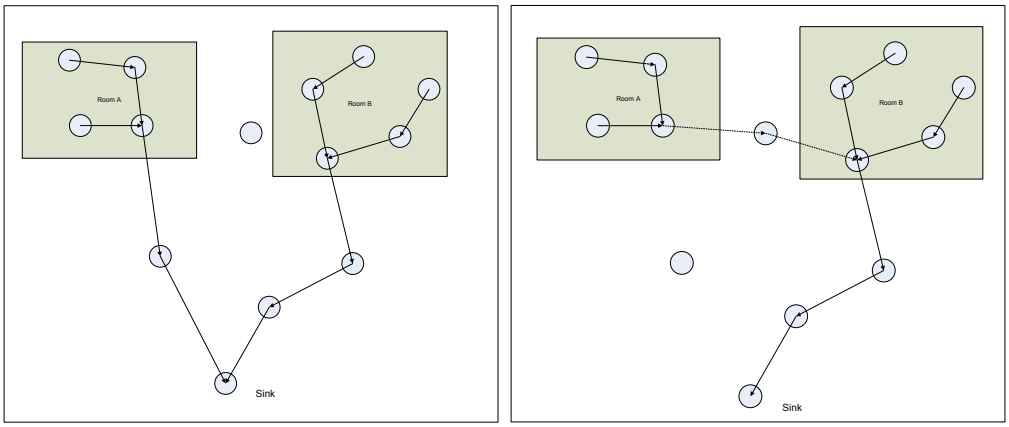
Figure 3.2: Type-based event model

ferent events, we can easily meet different application requirements without sacrificing the efficiency.

3.1.1 PSW-EDL: Event Definition Language in PSWare

PSWare provides a type-based event programming model to the application user. Such model has the following characteristics:

- Each event type is similar to a class in object-based programming model. Similarly, event hierarchy in Figure 1.3a is similar to class hierarchy.
- Similar to object-based model, attributes are encapsulated in each event.
- Event definition is declarative. The high-level application developers



(a) Centralized (b) Distributed

Figure 3.3: Motivating application for indoor monitoring

only need to specify the event filters and event relations through operators and functions. The underlying mechanisms for implementing these operators are left to our event processing framework.

Figure 3.2 conceptually shows our event model. On the top level, the application requirements are expressed in terms of the composite events that need to be detected. The composite events may be further divided into sub-events. Eventually all composite events can be divided into primitive events which can be directly detected by individual sensor nodes.

To illustrate using an example, consider a monitoring application shown in Figure 3.3. The composite event consists of two sub-events to be detected in two different rooms, Room A and Room B. It occurs when the temperature in room A rises to a certain threshold and after 5 minutes, the temperature in room B also reaches that threshold. We may define the events as shown in Listing 3.1.

Listing 3.1: Example of using event-based programming model

```
1  Event SimpleEvent {
2    int temp=System.temp;
3    int id=System.id;
4    int time=System.time;
5  } where {
6    temp > 30
7  }
8  Event CompEvent {
9  } on {
10   SimpleEvent e1 and
11   SimpleEvent e2
12 } where {
13   e1.Location="A" and
14   e2.Location="B" and
15   e2.time-e1.time=600
16 }
```

We can see that our event-based programming model shares some of the similarity with SQL, especially for the event filters, which consist of operators. In this particular example, we have two event types. On the top level, the application wants to monitor the temperature change so the event type 'CompEvent' is defined. 'CompEvent' consists of two sub-events of the same type: SimpleEvent. Note that our programming model is declarative. The user just specifies the event types and the corresponding filters without specifying the event processing methods because that part is left to the event processing framework. We will go into more details in that in the latter part of this section.

3.1.2 PSW-EN: Event Notifier in PSWare

Apart from submitting event definition, the application user needs to be notified when the subscribed events are detected by WSN. This is done through

event notifier. When the application user subscribes events, the user needs to pass an additional object as the event notifier. When the subscribed event is detected, PSWare will notify the user with this notifier object. Listing 3.2 shows our notifier class implemented and how it is related to the subscription class.

Listing 3.2: Event notifier in Java

```
1 public interface EventNotifier {
2   public void notify(String eventStr);
3   ...
4 }
5 public class EventSubscription {
6   public Boolean subscribe(String subscription,
        EventNotifier notifier) {
7     ...
8   }
9 }
```

While Listing 3.2 shows the notifier in Java, bindings for other languages can be created in similar fashion. A Python binding is shown in Listing 3.3.

Listing 3.3: Python binding of event notifier

```
1 class EventNotifier:
2   def notify(self, eventStr):
3     pass
4 class EventSubscription:
5   def subscribe(self, subscription, notifier):
6     ...
```

Upon the detection of events, the notifier will be invoked so that the events can be delivered to the user. The event is delivered as a string, with each attribute assigned with an actual value. As an example, suppose the user has subscribed to the event 'SimpleEvent' in Listing 3.1, then when the event is detected by PSWare, it will be delivered to the user with the content shown in Listing 3.4.

Listing 3.4: Received event from notifier

```
1 SimpleEvent e1 {
2   temp=32;
3   id=0;
4   time=13345;
5 }
```

3.1.3 API for Event Processing Framework

The event processing framework of PSWare is developed using NesC. PSWare also provides a group of APIs for the middleware developers to write customized event processing mechanisms. First, since all events can ultimately be decomposed into primitive events, the middleware developer needs to first define the required primitive events used in the application domain. This is done through a configuration named 'PrimitiveEventC'. It is listed in Listing 3.5. The new event ID should be defined as an enumeration in PSWare's header file.

Listing 3.5: Primitive event component in NesC

```
1 generic configuration PrimitiveEventC(evet_id_t eventId) {
```

```
2    provides {
3      interface Read<uint16_t>;
4    }
5  } implementation {
6    ...
7  }
```

In PSWare, everything is treated as an event and that includes temperature, photo and even timer. Similar to other generic components in TinyOS, if necessary, new events can be added by defining a new event ID. Once set, we need to create a higher level primitive event so that the application users can make new event definitions based on top of it. This is done by an automatic tool which will be invoked when building and output a class which will be used by the event notifier and an event definition which will be used by EDL.

When this is set, the application user can already start to subscribe and detect events by using PSWare's default event detection algorithm: TED [58]. If the middleware developer wants to write their own event detection algorithm, they can choose to implement two interface 'EventMatcher' and 'EventDeliver' as shown in Listing 3.6. The 'EventMatcher' interface includes two major events which serve three purposes as follows:

1. During the execution of the middleware, the network may detect multiple events for a single event type. Therefore, upon the detection of the composite events, the event detection algorithm may choose a specific event from one of this sub-types for detection. This is done by signaling the first event 'select sub-event'.

2. Upon the detection of an event, the event detection algorithm may perform some customized processing to update some information so that the next time when the event happens again, it may be detected with lower cost. This is when the second event 'eventMatched' comes into play. The middleware developer may implement customized event processing mechanisms according to the matched events.

Apart from event matcher, we also have an event deliverer which will be invoked when a subscribed event is detected. The middleware developer may implement its only function to meet the requirements for event delivery. This is done when the middleware signals the 'eventDeliver' event.

Listing 3.6: The event matcher interface

```
1 interface EventMatcher {
2   event bool selectSubevent(EventInstanceInfo * composite,
        EventInstanceInfo * subevent);
3   event result_t eventMatched(evet_id_t eventId, evet_id_t
        instanceID, bool detectionResult);
4 }
5 interface EventDeliverer {
6   event result_t eventDeliver(evet_id_t eventId, evet_id_t
        instanceID, bool detectionResult);
7 }
```

To facilitate the implementation of event matcher and event deliverer the middleware developer can make use of the APIs provided by PSWare in Listing 3.7. These APIs are mostly used to retrieve the event information.

Listing 3.7: PSWare API in NesC

```
1  interface EventType {
2    command EventTypeInfo * getEventType(evet_id_t eventId);
3    command bool isSubscribed(evet_id_t eventId);
4    command bool isComposite(evet_id_t eventId);
5    command EventRelation getRelation(event_id_t e1,
        event_id_t e2);
6  }
7  interface EventInstance {
8    command int instanceAmount(evet_id_t eventId);
9    command EventInstanceInfo * getEventInstance(evet_id_t
        eventId, evet_id_t idx);
10   command void deleteEvent(evet_id_t eventId, evet_id_t
        instanceID);
11 }
12
13 typedef struct {
14   evet_id_t eventId;
15   evet_id_t level;
16   size_t size;
17 } EventTypeInfo;
18
19 typedef struct {
20   evet_id_t typeID;
21   evet_id_t instanceID;
22   uint16_t * attributes;
23 } EventInstanceInfo;
```

The first interface, 'EventType' has four commands. The first command is for the individual event instances to obtain the type information based on their type ID as shown on line 20. The second and the third commands are used to determine if a given event type is a composite event or is subscribed by the user. The last command is for query the relation of e2 in reference to e1. Its return type is an enumerate which can have value of: parent, immediate parent, child and immediate child. This command will be useful if the user wants to implement customized event processing mechanisms.

Since for each event type, there can be multiple events, we need the second interface 'EventInstance' to process those events. There are three commands for this interface. The first one is used to obtain the number of the events for a specific type currently stored in the event buffer on the sensor node. We will discuss about the event buffer in the next section. Upon knowing the number of events, the middleware developer can iterate through the event list and use the second command to get the events. As for the last command, when an event is detected to be useless, the middleware developer may delete it from the list.

The data structures are shown after the interfaces. For a given event type, we have the ID for the type, its level in the subscribed event tree and its size. For the event instance data structure, we have its type ID, instance ID and its attribute list. If the event contains some attributes of the primitive events, they can be obtained here by using the index defined in the enumeration value for primitive event. For instance, if one of the primitive event has an enumeration value labeled 'EVENT_LIGHT', then if the event instance also has one of its attribute from the sensor's light reading, the middleware developer can access it by writing 'attributes[EVENT_LIGHT]'.

In summary, the middleware developer may follow the following steps to implement customized event processing mechanisms:

- Implement new primitive event types if necessary.
- Define the event matcher for how sub-events are selected for matching and what to do after a predefined event type is detected.
- Define customized function for event delivery.

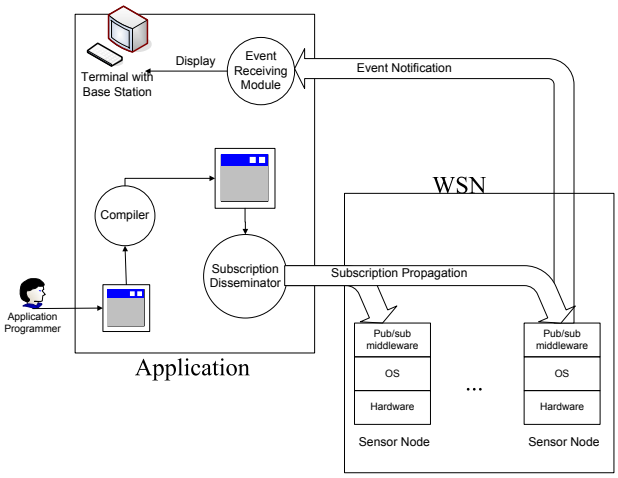


Figure 3.4: Event processing in PSWare

3.2 Composite Event Processing in PSWare

In this section, we discuss how PSWare is designed to support composite event. The overall work flow of event processing in PSWare is shown in Figure 3.4. To use the middleware, applications developers will first define event types according to the application requirements. The subscription will then be compiled and processed by the EDL compiler and be disseminated into the network. When the events are detected by sensor nodes, they will be delivered to the application.

3.2.1 Event Specification

EDL is used for specifying events in PSWare. For each EDL script, it contains one or more event definition and one subscribing statement. Formally, the Backus-Naur Form (BNF) of the subscription is defined in Listing 3.8.

Listing 3.8: BNF (simplified) of subscription

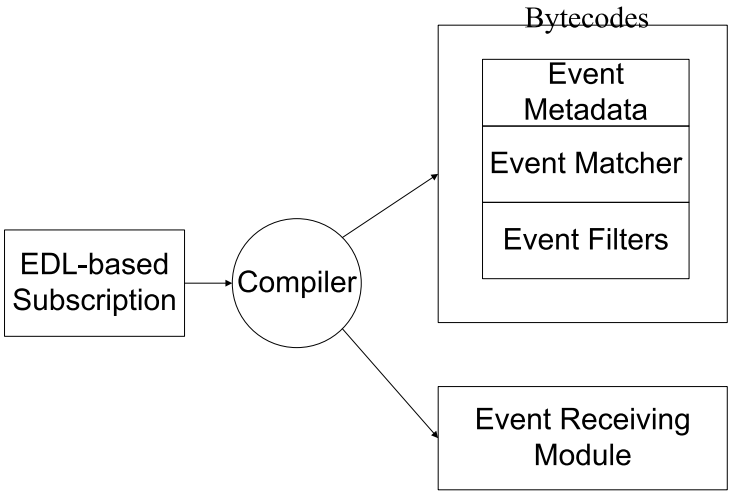


Figure 3.5: PSWare-EDL compiler structure

```
1  subscription -> event_declarations subscribe_statement
2  event_declarations -> event_declaration |
       event_declarations event_declaration
3  subscribe_statement -> SUBSCRIBE IDENTIFIER SEMICOLON
```

The subscribe statement simply uses the keyword 'subscribe' followed the event type name needed by the application. Each event type declaration can have up to three parts: the event body, the *where* clause and the *on* clause. The event body defines the attributes of the events. The *on* clause are used to specify the sub-events used by a composite event. The *where* clause defines the filter of the corresponding event type. Formally, the BNF of event type is defined in Listing 3.9

Listing 3.9: BNF (simplified) of event type

```
1  event_declaration -> EVENT IDENTIFIER event_body
       on_clause_opt where_clause_opt
2  event_body -> { field_declarations_opt }
```

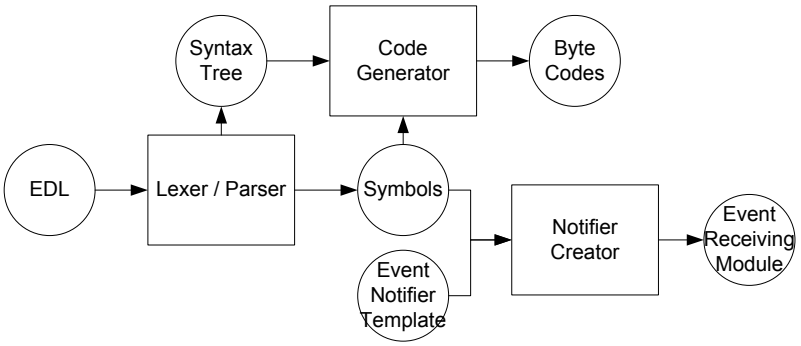


Figure 3.6: EDL compiler flow

```
3  on_clause -> ON { subevent_declarations_opt }
4  where_clause -> WHERE { conditional_expression }
```

The *on* clause and the *where* clause are both optional in case the event is primitive or does not have a filter. The on clause looks similar to the field declaration except sub-events instead of fields are declared. This is done for a clear code presentation and easier type checking. The where clause simply consists of conditional expressions so that the filters may be defined by specifying the operators.

The EDL-based subscription will be processed by our EDL compiler. The output of the compiler has two parts as shown in Figure 3.5. The first part is the byte codes which will be executed by individual sensors to detect events. The format and the organization of the byte codes are closely related to the event processing framework and customization of PSWare. We will go through these topics in the following sections.

The second part, the event receiving module is the implementation of the event notifier as discussed in Section 3.1. As shown in Figure 3.6, The EDL compiler will execute the following steps in order to generate the byte codes

and event receiving module:

1. Parse the EDL script and generate the corresponding syntax tree and symbol table.
2. Generate the byte codes based on the syntax tree and symbol table.
3. Create the event receiving module based on the symbol table.

3.2.2 Runtime Environment for Event Detection

The byte codes generated by the compiler can be further divided into three parts: event meta data, event filters and event matcher. These components implements the programming interface discussed in Listing 3.6 and 3.7. Event meta data contains the description of the event types such as event type ID, event size and the individual attributes for each event. Event filters are the constraints defined for each event type. Event matcher schedules the execution for event detection according to the subscription and event relations.

The runtime environment on each sensor node is similar to the VM-based approach [64] in the sense that subscriptions are broken down into some basic operations called instructions. For a complete list of instructions, please refer to the Appendix. Such design choice is for extensibility so that new features can be added more easily by adding new instructions. In addition to the VM-based runtime environment, each sensor node has an event buffer where the detected events can be stored for composite event detection.

The essential operations for our runtime environment are shown in Figure 3.7. In this environment, the event matcher will first fetch the events from

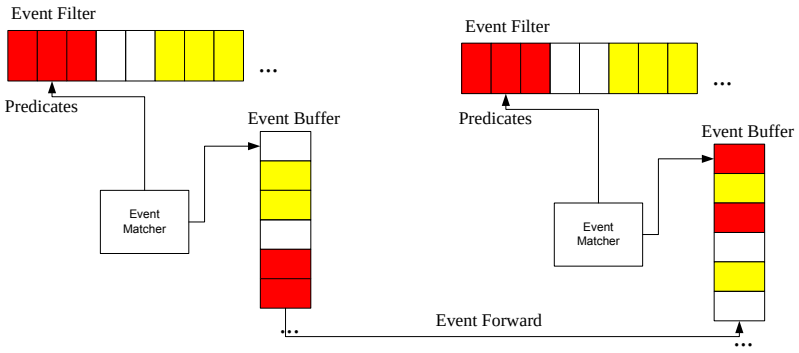


Figure 3.7: PSWare runtime environment

the event buffer and then evaluate them against the corresponding filters. If the event has been detected, then it will be transmitted over the network. Formally, the procedure of the event matcher can be shown in Procedure 1 with some notations defined as:

- Event types: $E = \{e_1, e_2 \cdots\}$
- For each $e_n \in E$, its filter is: $e_n \rightarrow filter$
- For each $e_n \in E$, it has a set of events $E_n = \{e_n^1, e_n^2 \cdots\}$ stored in the buffer.

There are several keys in the procedure. First, when the event matcher picks up the events of type e_n from the event buffer, it may use application specific mechanisms to pick up the desired events instead of trying all the possible combinations. Second, the 'deliver()' and the 'forward()' function are used to deliver the subscribed events or forward the events so that composite events may be detected. These two functions may also be application dependent to achieve high energy efficiency.

Finally, it is necessary to mention the 'SystemEvent'. This module acts

Procedure 1 Procedure of the event matcher

Input: E

```
for all e_n ∈ E do
  for all e_n^i ∈ E_n do
    if e_n is primitive then
      result = evaluate_primitive (e_n^i)
      if result == True then
        eventMatched(e_n^i)
        if e_n is subscribed then
          deliver(e_n^i)
        else
          forward(e_n^i)
        end if
      end if
    else
      e_sub = ∅
      for all subevents e_m for e_n do
        e_sub = e_sub ⋃ selectSubevent(e_n, e_m)
      end for
      for all subevents e_m for e_sub do
        evaluate_composite(e_n^i, e_m, ···)
      end for
    end if
  end for
end for
```

as the device driver for PSWare. It defines the sampling rate and a primitive event. All the fields of other events are obtained from 'System'. The module needs to implement three interfaces: StdControl, SystemClock and SystemEvent as shown in Listing 3.10. StdControl is a module for initialization purpose. SystemClock defines the sampling frequency. SystemEvent is used to obtain the pointer to the 'System' event.

Listing 3.10: API of the 'System' event

```
1  module SystemEventM {
2    provides {
3      interface StdControl;
4      interface SystemEvent;
5      interface SystemClock;
6    }
7  }
8  interface SystemEvent {
9    command EventInstanceInfo * get();
10 }
```

The 'SystemEvent' is there so that needs to be implemented by the middleware developers as the Once the 'System' event is defined, the application developers can further define their own functions for event delivery and event forwarding. We will show some examples in the next section.

3.3 Support for Customization in PSWare

An important feature of PSWare is that it can be customized. To support customization, PSWare uses a flexible layered architecture. In this way,

developers can customize different layers without affecting each other. Moreover, multiple event processing strategies may be dynamically used during the runtime. In this section, we how such customization can be done at different layers. These layers implement different aspects of a complete event-based system, including, event detection, event delivery and event subscription.

3.3.1 Customizable Event Definition

It is a common scenario for applications to define extra event attributes that have domain-specific meanings. For example, in an application which requires reliable communication, we might want to define a probability value which specifies the threshold for message loss. Then, in the middleware framework, this number should be used during the actual communication.

The simplest way to pass some domain-specific information to the middleware is to modify the System event. As discussed in Section 3.2, the System event is used like a device driver that represents the primitive events collected by the system. Internally, this event is specified in a header file and can be modified to suit different applications. The essential steps are as follows:

1. Modify 'SystemEvent.h' and add necessary attributes for the system event

2. Use 'psware gen' to generate the necessary constant values for accessing the new attribute in the middleware runtime environment

3. Modify the middleware framework so that the attributes can be used

4. Use the new attributes in the actual event definition

We will illustrate these steps through an application scenario. Suppose the middleware developer has implemented a message retransmission mechanism for the event delivery and matching functions to make the application more reliable. Then the application developers can specify a parameter indicating the desired reliability for the events they define. The new 'SystemEvent.h' will look like in Listing 3.11.

Listing 3.11: Customized system event

```
1 typedef struct {
2   uint16_t nodeID;
3   uint16_t time;
4   float probability;
5 } SystemEvent;
```

Once the new probability attribute is defined, we need to generate some necessary constant values for accessing the attribute. The tool for generating the constants is 'psware gen'. It will parse the event header files and output some macro values. After that, the middleware can access the attribute with the code fragment shown in Listing 3.12. In this piece of code, we first obtain the system event through the EventInstance API. Then we obtain the probability value by accessing the correct attribute. Note that SystemEvent, SystemEvent_probability are generated constant values for accessing the events and their attributes.

The event probability can then be further defined through event definition. To reuse the event definition in Listing 3.1, now if the user wants to

Listing 3.12: Customized system event

```
1 EventInstanceInfo * systemPtr = call EventInstance.
    getEventInstance(SystemEvent);
2 float probability = (float)systemPtr->content[
    SystemEvent_probability];
```

add a parameter to indicate the reliability, he may insert a statement at Line 5 which defines the global reliability parameter.

Listing 3.13: Example of event definition with reliability

```
1  Event SimpleEvent {
2    int temp=System.temp;
3    int id=System.id;
4    int time=System.time;
5    System.reliability = 1.0;
6  } where {
7    temp > 30
8  }
9  Event CompEvent {
10 } on {
11   SimpleEvent e1 and
12   SimpleEvent e2
13 } where {
14   e1.Location="A" and
15   e2.Location="B" and
16   e2.time-e1.time=600
17 }
```

Once the parameter is defined. The information will be passed to the middleware and the corresponding code fragment in Listing 3.12 will work

as expected.

3.3.2 Customizable Event Detection

Event detection is the heart of an event-based system. In PSWare, we can easily customize event detection on the event detection layer. Since PSWare uses a type-based event model to support composite event, event detection can also be customized according to the event types. The customization takes the following steps:

1. Define some domain-specific event types that requires specific event detection methods

2. Generate the necessary constant values for accessing the new attribute in the middleware runtime environment

3. Implement the specific event detection methods in the middleware

We demonstrate the steps through a simple example, iTED. iTED is a simplified version of the TED [58] algorithm, the default event detection algorithm in PSWare. We will discuss our default event detection algorithm, TED, in the next chapter. Different from TED, iTED is customized for indoor monitoring applications where the events are first fused in each monitored room. Then, the results are further fused for global event detection. For simplicity, we assume each sensor node is equipped with a room ID and there maybe more than one fusion points in each room. However, all fusion points are selected in advance and will remain unchanged. The room ID can either be pre-deployed in the sensor nodes' program or be obtained using

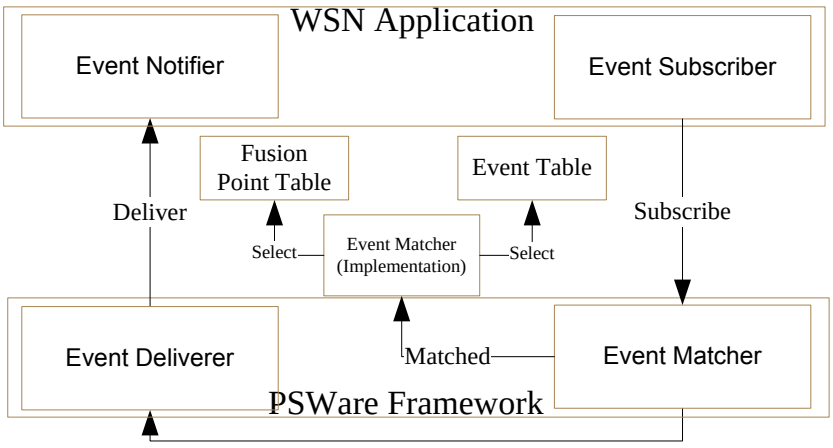


Figure 3.8: iTED over PSWare

localization methods and more sophisticated fusion point selection methods may be implemented by extending some components discussed here.

Figure 3.8 shows an overall diagram on how iTED interacts with PSWare. To implement iTED in PSWare, we need three key components:

1. Fusion point table: each sensor node updates this table in order to find the fusion point.

2. Event table: each sensor node maintains this table in order to decide if a given detected event should be forwarded to the fusion point.

3. Event matcher: a component that implements the event matching interface as discussed in Section 3.1, Listing 3.6.

The first step is to add a new attribute, roomID, to our system event. This has already been discussed in the previous sub-section in Listing 3.11 and we will skip it here. The only difference is the new attribute here will be of 'int' type instead of 'float' type.

Our first component, the fusion point table maintained by each sensor node $v'_n \in V'$ is denoted as $table_r$. It contains the following data:

- Hop count (hop_n): the number of hops to reach the fusion point
- Parent ($parent_n$): the next hop to that fusion point

The procedure for updating fusion point table is shown in Procedure 2. Note that we make use of the roomID to filter the messages.

Procedure 2 Fusion point table exchange

Input: $v_n \rightarrow msg_r$
  for each entry t' in msg_r **do**
    if not exists $t' \rightarrow fid_n$ in $table_r$ **then**
      $addTo(table_r, t')$
    end if
    for each entry t in $table_r$ **do**
      if $t \rightarrow roomID = t' \rightarrow roomID$ **then**
        if $t' \rightarrow hop_n < t \rightarrow hop_n$ **then**
          $t \rightarrow hop_n \leftarrow t \rightarrow hop_n + 1$
          $t \rightarrow parent_n \leftarrow v_n$
        end if
      end if
    end for
  end for
  if self is fusion point **and not** exists $self \rightarrow id$ in $table_r$ **then**
    $addTo(table_r, (self \rightarrow id, 0, self \rightarrow id))$
  end if
  $msg_r \leftarrow table_r$
  $periodically\_broadcast(msg_r)$

The second and the last component, the event table and the event matcher are closely related. Each sensor node will maintain an event table which is denoted as $table_e$. The event table contains information for each event type $e_n \in E$ as follows:

- Event type ID (e_n): the ID which is assigned to each event type
- Fusion point for the event ($fusion_n$): the fusion point at which the event is mostly likely to be detected at the lowest cost.
- Fusion cost ($cost_n$): the fusion cost for event type e_n

In addition, each fusion point $v' \in V'$ will maintain another table, the event matching $table_m$ for the purpose of matching events. $table_m$ contains the following fields:

- Event type ID (e_n): the ID of the event type
- Event instance ID (i): the ith event instance of event type e_n (we use e_n^i to denote such an instance of event)
- Source node (v_n^i): the node which forwarded e_n^i to the fusion point
- Event timestamp (t_n^i): the timestamp when the event e_n^i is detected
- Detection cost ($cost_n^i$): the cost for detecting event e_n^i

First, each node v_n periodically broadcasts messages msg_r which is its $table_r$. If the node itself is a fusion point, then it will add itself in $table_r$ and broadcast the message. The procedure is shown in Procedure 2. In addition to msg_r, each $v'_k \in V'$ will periodically advertise its $table_m$ by broadcasting msg_m so that other sensor nodes can construct their $table_e$ with Procedure 3.

The construction of $table_m$ will take place when the event instance e_n^i is detected and forwarded to a fusion point v'_n. We will discuss how forwarding could be done in the next subsection.

Procedure 3 Event table exchange

Input: $v'_k \rightarrow msg_m$
  for each entry t' in msg_m **do**
    if not exists $t' \rightarrow e_n$ in $table_e$ **then**
      $addTo(table_e, (t' \rightarrow e_n, 1, v'_k, t' \rightarrow cost^i_n + table_r \rightarrow hop_k))$
    end if
    for each entry t in $table_e$ **do**
      if $t \rightarrow e_n = t' \rightarrow e_n$ **then**
        if $t' \rightarrow cost^i_n + table_r \rightarrow hop_k < t \rightarrow cost_n$ **then**
          $t \rightarrow cost_n \leftarrow t' \rightarrow cost^i_n + table_r \rightarrow hop_k$
          $t \rightarrow fusion_n \leftarrow v'_k$
        end if
      end if
    end for
  end for
  if self is fusion point **then**
    $msg_m \leftarrow table_m$
    $periodically\_broadcast(msg_m)$
  end if

When an event e^i_n is matched at node v_k, node will use $table_r$ and $table_e$ to decide how to forward the detected event to the fusion points so that higher level events can be matched. In case the fusion point for event type e_n has not been decided, the node will forward the event to some of its closest fusion points according to iTED. Upon the reception of e^i_n from v_k, the fusion point will first update its own $table_m$. Then it will check if there is any composite event e_{comp} which uses e_n and another event e_j as its sub-event ($e_{comp} = comp(e_n, e_j)$). If there is, then e_j will be used upon 'selectSubevent'.

If e_{comp} has been successfully detected by the underlying event matcher, then Procedure 4 will be executed so that $table_m$ is updated accordingly and iTED may reduce the energy cost for future event detection. Note that Procedure 4 will make use of the APIs in Listing 3.7 since it needs to query

Procedure 4 Event matching

Input: e_n^i matched by v_k with cost: $cost_n^i$
  $addTo(table_m, (e_n, e_n^i, v_k, now(), cost_n^i))$
  for each e_j in E **do**
    if $\exists r \in R$ **and** $r = e_{comp} = comp(e_n, e_j)$ **then**
      for each e_j^k in $table_m$ **do**
        if $comp(e_n^i, e_j^k) = true$ **then**
          $addTo(table_m, (e_{comp}, e_{comp}^i, self, now(), cost_n^i + cost_j^k))$
          $detected(e_{comp})$
        end if
      end for
    end if
  end for

the event relations.

3.3.3 Customizable Event Delivery

After the subscribed event is detected, it needs to be delivered to the user. In many applications, this is done via the underlying routing protocols such as Collection Tree Protocol (CTP) [33] provided by TinyOS. This is also the case for the default event delivery in PSWare. However, in some applications, this may not be a case. For example, in an intelligent transportation system, the events may be delivered to mobile vehicles. In this subsection, we show how PSWare can achieve the flexibility in event delivery through several examples in ITS.

We choose ITS as an example because different event types in this application may require different event delivery strategies and that is why the flexibility in event delivery is of particular importance. The event types in ITS may include:

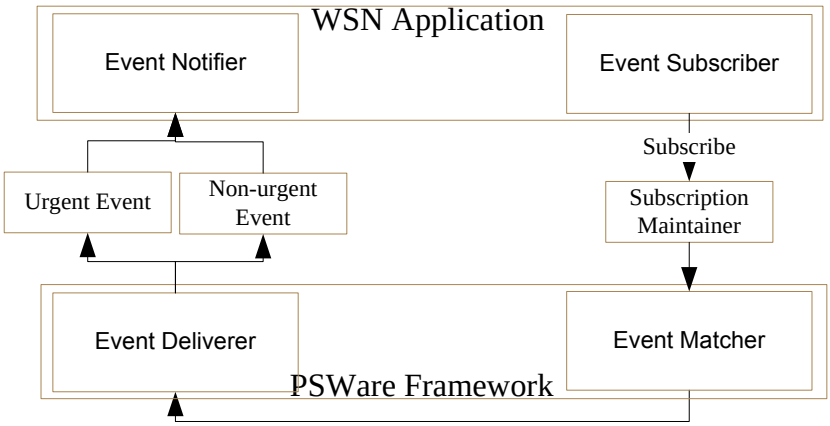


Figure 3.9: Type-based event delivery

- Emergency: events that represent urgent incidents. Examples of this type of events include car accidents and urgent road maintenance. These events probably need to be delivered to all nearby vehicles when occurred.

- Driver's information: events that provide assistant information to the drivers. Examples include congestion information and meteorological information. These events are usually not considered as urgent and may be delivered in carry-and-forward fashion [47].

The overall architecture for ITS event delivery over PSWare is shown in Figure 3.9. A key difference in this architecture is the introduction of multiple event deliverer. Upon the signaling of 'eventDeliver' in Listing 3.6, the middleware developer can first query the event type by using the APIs in Listing 3.7. With 'psware gen', this may be easily achieved. The middlewre just needs to pre-define a domain-specific event type for traffic accident. When events of such types are detected, an emergency delivery method is used. Otherwise, the normal event delivery method will be used.

Chapter 4

Generic Composite Event Detection

4.1 The Composite Event Detection Problem

In previous sections, we have discussed the overall design for PSWare. Once application programmers define the composite event types and make subscriptions through our middleware, the sensor nodes need to process the data and detect the subscribed composite events. In fact, a generic event detection algorithm is the heart of a pub/sub system. In this section, we formally define our generic composite event detection algorithm used by PSWare.

4.1.1 System Model

We consider the network as a graph $G = (N, A)$ where each node represents a sensor node and each edge represents a communication link. For each $a_n \in A$, it has a weight W_n associated with it.

The subscriber provides a finite set of event types $E = \{e_1, e_2, \cdots\}$. For each $e_n \in E$, the subscriber defines a set of attributes $e_n \rightarrow attr_n$ which reflect certain real world phenomenon.

The subscriber also provides a finite set of event relations $R = \{r_1, r_2, \cdots\}$ where each $r_n \in R$ represents the mapping of one or more sub-event types $e_1, e_2 \cdots \in E$ to a composite event type $e_3 \in E$, denoted as $r_n(e_1, e_2, \cdots) = e_3$. One of the event type $e_s \in E$ is subscribed by the subscriber.

We have a set of primitive event types $E_{primitive} \subseteq E$ event types such that there is no $\exists r_n \in R$ such that $r_n(e_1, e_2, \cdots) = e_n$ where $e_n \in E_{primitive}$ and $e_1, e_2, \cdots \in E$. For each primitive event of type $e'_n \in E_{primitive}$, it will be detected by a node $n_i \in N$. For the sake of simplicity in discussion, we only consider the message cost in our energy cost function and we do not consider message retransmission. The event detection cost for each event type $e_n \in E$ is denoted as $cost(e_n)$. Such cost is the number of hops for the event to be delivered from the event source to its destination. Such destination could be an event fusion point (to be discussed soon) for detecting higher level composite events or the sink node where the events will be delivered.

4.1.2 Problem Formulation

Given:

- A network $G = (N, A)$
- A set of event types E with relation R
- An energy cost function $cost(e_n)$ for $e_n \in E$

Find:

- For each event type $e_n \in E$, when an event instance of this type occurs, find a subset of nodes $V_n^r \subseteq V$ which are involved in detecting the event.

Objective:

- Minimize the message cost:

$$\sum_{i=i}^{n} cost(e_i)$$

Theorem 1. *The composite event detection problem is NP-complete.*

Proof. We show our proof by reducing the steiner tree problem to our composite event detection problem. Let $N_s \subset N$ be the event source (nodes that detect the primitive events). Since the cost is defined as message cost, if we minimize the total path length from the event sources to the fusion points (nodes which are responsible for detecting composite events), then we can also minimize the message cost.

We construct a graph $G' = (N', A')$ from G as follows:

1. $N' = N_s \bigcup N_f$
2. For each pair of nodes $n'_i, n'_j \in N'$, we add an edge $a'_k \in A'$ incident on both if there is a path from n'_i to n'_j in G.
3. The weight of the newly added edge is a'_k is the weight of the shortest path from n'_i to n'_j in G.

The corresponding Steiner tree problem can be defined as below.

Given:

- A graph: $G' = (N', A')$
- Each edge e'_i in the graph has a weight of W'_i
- A set of sources: $N_s \subset N$

Find:

- A minimum Steiner tree that spans N_s

If we have a solution for the Steiner tree problem in G', then we simply need to recover the shortest paths in G and it will also be the optimal solution for our composite event detection problem. On the other hand, an optimal solution for our composite event detection problem is also an optimal solution for the Steiner tree problem if we replace the paths between every pair of nodes $n'_i, n'_j \in N'$ with the edges in A'. □

4.2 A Centralized Approach

Based on our problem formulation, we can intuitively use a centralized approach to solve this problem. In the centralized approach, when the sensor nodes detect the events, they first inform the sink node about the detected event. Then the sink node will select the efficient nodes as event fusion points for each event type and broadcast the information into the network. The nodes will then forward their events to these fusion points. The fusion points may be changed or updated with a predetermined probability to cope with event dynamics. The reasons why the nodes will only inform the sink about the events they detected rather than send the events to the sink for detection are as follows:

- Each individual event may have a lot of attributes so it is more desirable to first calculate the energy-efficient event fusion points before the actual event detection.

- As described in Section 1.1, events may have strong locality. Therefore, the fusion points selected may be used for detecting events of the same types in the future.

We assume the users have no prior knowledge on where the events might happen. However, they do know the probability distribution of the events. We will discuss the event probability distribution and its impact on the algorithm in the latter section when we describe how the algorithm is designed to cope with event dynamics.

The input of the algorithm includes:

- Sensor network: G
- A set of events $E' = \{e_i^1, e_i^2, \cdots\}$ where e_i^j denotes an event of type e_i

Algorithm 5 Centralized TED

Input: $G = (N, A)$, E'

1: **for all** $e_i^j \in E'$ **do**
2: **if** e_i is not assigned to any node **then**
3: Find any other $e_k^l \in E'$ such that either k==i or e_k and e_i has a relation for a composite event e_x
4: **for all** $n \in N$ **do**
5: Construct an SPT for all e_k^l and e_i^j with root as n
6: Calculate the cost for the SPT as $cost_n$
7: **end for**
8: Find the SPT with the smallest $cost_n$ and select n as fusion point for e_k and e_i
9: Add n as the node detecting e_x^1 to E'
10: **end if**
11: **end for**

The algorithm will examine all the primitive events reported from the network to see if there is any relation associated with them. If so, the algorithm will try each node in G to construct shortest path tree (SPT) for these related events and select the one with the smallest cost. The root of that SPT will then become the fusion point for the related events. The algorithm runs recursively in the sense that once the fusion points for a composite event is selected, itself will become the node for detecting that composite event.

4.2.1 Determine the Re-selection Probability

While the previous section outlines the algorithm for centralized TED, we still need to decide how often the nodes should switch to another fusion point in order to cope with the event dynamics. We use exponential distribution as

the event probability distribution because of its memoryless property. More specifically, for each composite event, the distance x between each of its sub-events to any point in the network follows an exponential distribution as follows:

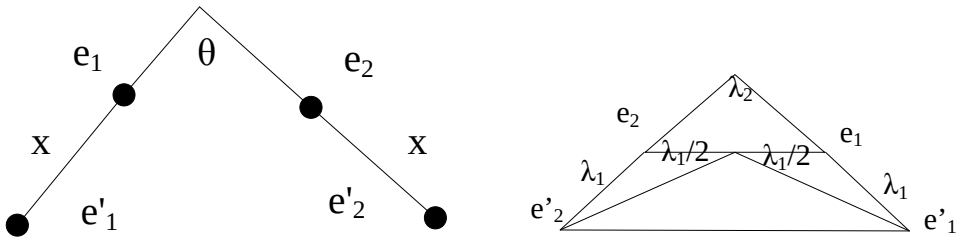
$$f(x) = \lambda_1 e^{-\lambda_1 x}$$

In addition to the distance between events, the direction of events that happen in different rounds will also affect the selection probability. The angle θ between any pair of related events also satisfy an exponential distribution as follows:

$$f(\theta) = \lambda_2 e^{-\lambda_2 \theta}$$

Both x and θ are shown in Figure 4.1b where the events e_1, e_2 are detected before and e_1', e_2' are detected. For simplicity, we use distance to measure the cost for one node to reach another. As shown in Figure 4.1b, for a composite event that has two sub-events, originally the event is fused at n_1. Then, for the next detection of e_1' and e_2', if the fusion point is still the original one, then the cost will be the added distance of (e_1', n_1) and (e_2', n_1). To calculate such cost, we first need to calculate the angle θ_1 between (e_2', e_2) and (e_2, n_1):

$$\begin{aligned}\theta_1 =& \pi - \frac{\pi - \lambda_2}{2} \\ =& \frac{\pi + \lambda_2}{2}\end{aligned}$$



(a) Event distribution (b) Fusion point distance

Figure 4.1: Selecting fusion points in centralized approach

Then according to law of cosines, we can derive $cost_1$:

$$\begin{aligned} cost_1 =& 2 \times \sqrt{(\lambda_1)^2 + (\frac{\lambda_1}{2})^2 - \lambda_1 \times \frac{\lambda_1}{2} cos(\frac{\pi + \lambda_2}{2})} \\ =& 2 \times \sqrt{\frac{5(\lambda_1)^2}{4} - (\lambda_1)^2 cos(\frac{\pi + \lambda_2}{2})} \end{aligned}$$

On the other hand, if a new fusion point is selected, according to law of sines, then the cost will be no more than:

$$\begin{aligned} cost_2 =& 2(\frac{\lambda_1}{2sin\frac{\lambda_2}{2}} + \lambda_1) sin\frac{\lambda_2}{2} + d \\ =& \lambda_1(1 + 2sin\frac{\lambda_2}{2}) + d \end{aligned}$$

Here, d is the average distance between any point in the deployment region to the closest fusion point.

The condition to select a new fusion point will be:

$$
\begin{aligned}
& cost_1 \geq cost_2 \\
& cost_1^2 \geq cost_2^2 \\
& (\lambda_1)^2(\frac{5}{4} - cos(\frac{\pi + \lambda_2}{2})) \geq \\
& \lambda_1^2(1 + 2sin\frac{\lambda_2}{2})^2 + d^2 + 2\lambda_1(1 + 2sin\frac{\lambda_2}{2})d
\end{aligned} \tag{4.1}
$$

Here d is decided by the node density and can be estimated once we know the deployment area and the number of nodes in the deployment area. Therefore, given $f(x)$, $f(\theta)$ and d, we can use generalized gradient search to find the values for θ and x such that Equation 4.1 is satisfied while minimizing the probability for switching:

$$
\begin{aligned}
P_{switch} &= F(x > x*)F(\theta > \theta*) \\
&= \int_{x*}^{\infty} \lambda_1 e^{-\lambda_1 x} dx \int_{\theta*}^{\infty} \lambda_2 e^{-\lambda_2 \theta} d\theta \\
&= e^{-\lambda_1 x*} e^{-\lambda_2 \theta*}
\end{aligned}
$$

Once the probability is obtained, after each event detection, there will be a probability of P_{switch} that the event will switch to another fusion point. This is done by making the fusion point broadcast a message in the network so that all nodes can delete the corresponding event type assignment to that fusion point.

So far the calculation for P_{switch} is based on the assumption that the network scale is not known in advance and the cost for broadcasting new

fusion point information is not known in advance. If the network scale is known to be $|N|$ in advance, then we can change Equation 4.1 a bit to be more accurate as in Equation 4.2.

$$cost_1 \geq cost_2 + |N| \times P_{switch} \tag{4.2}$$

By calculating $|N| \times P_{switch}$, we can further fine-tune the switching probability to balance with the broadcast cost.

4.3 TED: a Type-based Event Detection Algorithm

In this section, we propose TED, a distributed type-based composite event detection algorithm for WSN. The essential idea of TED is that after each sub-event is detected, the nodes will at first forward the detected events randomly to some nearby fusion points in the hope that at least some of them will be good ones. When the composite events are detected, the fusion points will first check the record to see if the source nodes have already selected any fusion point. If not, it will flood some feedback in the network so that the source node will get it and other nodes can also use such feedbacks as 'hints' when they need to forward the events. By collecting different feedbacks from different fusion points, the sensor nodes will choose the best one according to the cost. If the sub-events occur again, the nodes will be able to forward the detected events based on the feedback so that the cost could likely be

reduced.

4.3.1 Algorithm Input

In TED, the set of event fusion points $N_f \subseteq N$ are preselected. We will discuss how to select the fusion points in an optimal way in the latter part of the section. Therefore, each node will play two possible roles: normal node or event fusion point. Normal nodes will need the following data structure for the algorithm:

- Event filter table $table_f$: this table stores the filters for each event type. $table_f \rightarrow filter_n$ denotes the filter for event type e_n.
- Fusion point routing table ($table_r$): this table defines the routing to each fusion point $n_i \in N_f$. $table_r \rightarrow n_i \rightarrow parent$ denotes the parent node to reach fusion point n_i.
- Event forwarding table ($table_e$): this table defines the fusion point for each event type $e_n \in E$. $table_e \rightarrow e_n \rightarrow fp$ denotes the fusion point for event type e_n.

The fusion points will also have the same data structure of the normal nodes for the algorithm. In addition, they will have an additional table $table_m$. This table temporarily stores the events collected from other nodes. For each of the entries it has the following contents:

- e_n^i: the i^{th} event of type e_n
- *cost*: the detection cost for the event e_n^i

- $flag$: the flag (to be described in the algorithms) for the event e_n^i

4.3.2 TED for Normal Nodes

Since the event detection starts from primitive events, the normal nodes will run Algorithm 6 after detecting a primitive event e_n^i of type e_n. For each event type, it has three possible states: fpUnknown, fpIndicated and

Algorithm 6 TED for normal nodes

Input: evaluate(e_n^i, $table_f \rightarrow filter_n$)==True
1: **if** $table_e \rightarrow e_n \rightarrow flag \neq fpUnknown$ **then**
2: $toForward = table_e \rightarrow e_n \rightarrow fp$
3: Set $e_n^i \rightarrow flag = table_e \rightarrow e_n \rightarrow flag$
4: Forward e_n^i to $table_r \rightarrow fp \rightarrow parent$
5: **else**
6: **if** $table_e \rightarrow e_n \rightarrow flag \neq fpUnknown$ **then**
7: Select $k-1$ nearest fusion points $N_k \in N_f$
8: $N_k = N_k \bigcup \{table_e \rightarrow e_n \rightarrow fp\}$
9: **else**
10: Select k nearest fusion points $N_k \in N_f$
11: **end if**
12: **for** each $n \in N_k$ **do**
13: forward e_n^i to $table_r \rightarrow n \rightarrow parent$
14: **end for**
15: **end if**
16: **if** $e_n^i \rightarrow flag = fpSelected$ **and** $e_n^i \rightarrow timeout == True$ **then**
17: $table_e \rightarrow e_n \rightarrow flag = fpIndicated$
18: **end if**
Input: feedback of event type e_n from $n_i \in N_f$
19: $entry = table_e \rightarrow e_n$
20: **if** $e_n \rightarrow source == self$ **and** ($entry \rightarrow flag \neq fpSelected$ **or** $entry \rightarrow flag == fpSelected$ **and** $entry \rightarrow cost < e_n \rightarrow cost$) **then**
21: $entry \rightarrow flag = fpSelected$
22: $entry \rightarrow fp = n_i$
23: **else if** $entry \rightarrow cost < e_n \rightarrow cost$ **then**
24: $table_e \rightarrow e_n \rightarrow flag = fpIndicated$
25: **end if**

fpSelected. Initially, all the event types are fpUnknown because the sensor node does not know which fusion point is the best to forward the event. The flag will be updated upon the reception of feedbacks from the fusion points. More specifically, if the event is detected at the fusion point n_i, the fusion point will flood the feedback with cost and event source included so that the nodes can update their corresponding flags. The update is based on the detection cost.

Upon the detection of event e_n^i, the node will first check if there is already a fusion point assigned to it. If so, the event will simply be forwarded to that fusion point. Otherwise, the node will choose k closest fusion points randomly and then forward the events to them.

4.3.3 TED for Event Fusion Points

When the fusion point receives e_n^i from a node, it will first wait a period of time until the expiry time of the event to check for other events for possible matches. If no match is found during this period, the fusion point will still use Algorithm 6 to further forward the events to other fusion points. The pseudo code is shown in Algorithm 7.

The function 'detected' is the place where Algorithm 6 is invoked. Upon the detection of any composite event, the fusion point will also send the feedbacks to the network.

Algorithm 7 TED for fusion points

Input: e_n^i from node $n_i \in N$
1: **for all** $e_j \in E$ **do**
2: **if** e_n is a subevent of e_j **then**
3: result = evaluate e_j with e_n^i
4: **if** result==True **then**
5: detected (e_j)
6: $e_j \rightarrow cost = e_j \rightarrow cost + e_n^i \rightarrow cost$
7: $e_j \rightarrow source = e_j \rightarrow source \bigcup e_n^i \rightarrow source$
8: feedback (e_j)
9: **end if**
10: **end if**
11: **end for**
Input: expiry time of e_n^i
12: detected (e_j)

4.4 Fusion Point Deployment Problem

Because our distributed algorithm is based on certain nodes in the network that acts as event fusion points to detect the events, in this section, we discuss how to select such fusion points in order to optimally detect the events.

4.4.1 Even Deployment

We first look at a deployment where fusion points are evenly deployed. We use the following deployment model:

- The entire network is divided into a set of equally sized regions.
- Within each region, we deploy the same number of event fusion points.

Such deployment model is suitable if the user has no prior knowledge on where events would happen. After calculating the optimal deployment strategy, the users can make use of it in two ways:

- After the sensor deployment, the users can deploy additional sensor nodes as fusion points in the network.
- Before the deployment, the user can calculate how many fusion points are needed in the network and mix them with normal nodes to deploy them randomly.

We will use square for calculating the optimal deployment strategy in this work. The optimal deployment strategies with regions of other types of shape may be also be obtained in a similar fashion. Suppose we divide the whole region of area A into squares of size $s \times s$. Then on average, each sensor node can find a fusion point at a distance of [90]:

$$\begin{aligned} r =& \int_0^1 D(t)dt \\ =& \int_0^1 \frac{2}{3}\sqrt{c^2t^2 + (b^2 - a^2 - c^2)t + a^2}dt \\ =& \frac{c}{6}[u(1+v^2) + \frac{1}{2}(1-u^2)(1-v^2)ln(\frac{u-1}{u+1})] \end{aligned} \tag{4.3}$$

where,

$$\begin{aligned} c =& \frac{s}{2} \\ u =& \frac{\sqrt{2}+1}{2}s \\ v =& \frac{\sqrt{2}-1}{2}s \end{aligned} \tag{4.4}$$

In order to determine the optimal deployment strategy, we also need to know the event probabilistic distribution. We use the same exponential distribution model as introduced in Section 4.2.

The cost introduced by TED mainly consists of three parts: forwarding cost, feedback cost and detection cost. Initially, upon the detection of primitive events, the nodes will randomly forward the events to k closest fusion points.

$$cost_{forward} = r \times k$$

Here k is determined by the event distribution such that after forwarding different sub-events to the fusion points, there will be some overlapping fusion points for different sub-events. Therefore, k is defined as follows:

$$k = (\frac{\lambda_1}{r} + 1)^2 \tag{4.5}$$

When the events are detected at the fusion points, feedback will be sent to the event sources so that the sensor nodes can later forward the events to them and the cost will be reduced. For simplicity of analysis, we assume the fusion points will simply flood the feedback in the network. Therefore, the feedback cost is:

$$cost_{feedback} = |N| \times k$$

The detection cost is the message cost for all sub-events to be forwarded to a fusion point so that the composite event may be detected. As shown in Figure 4.2, if we have two events e_1 and e_2, the minimum event detection cost will be detecting the events on the line segment that connects the two events. However, we may not find a fusion point on the line segment, so in order to find a fusion point that can minimize energy cost, we should choose a point that lies on the center of the line segment. Similar to Equation 4.3

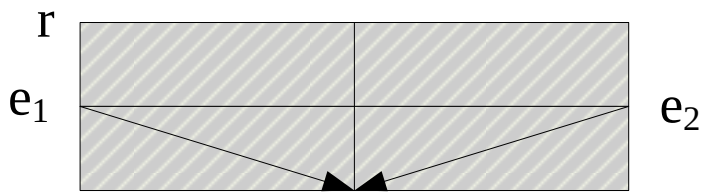


Figure 4.2: Event detection cost

and 4.4, the average detection cost will be:

$$cost_{detect} = 2 \times \int_0^{arctan\frac{2r}{\lambda_1}} \frac{\lambda_1}{2cosx} dx$$

Since each node needs to know how to reach the fusion points when forwarding is needed, there is overhead for maintaining such information. Similar to many existing routing protocols for WSN, we assume the nodes will periodically send messages for link evaluation [101]. Therefore, the cost for maintenance is:

$$cost_{maintenance} = (\frac{A}{s^2})|N|c_1$$

Here, c_1 is constant that represents the relation between energy consumption and the size of the packets. In addition, the sensor node should also have storage constraint because the nodes simply might not able to store all the routes to every fusion point. The storage constraint is defined as:

$$(\frac{A}{s^2}) < c_2$$

Objective is to minimize:

$$cost_{all} = 2(\frac{T}{t} + 1)cost_{forward} + cost_{maintenance} + Tcost_{detect}$$

λ_1	Expected location between the events
λ_2	Expected angle of the events
A	Deployment area
$s \times s$	The size of the square sub-regions
c_1	Energy cost per bit of data transmission
c_2	Storage constraint

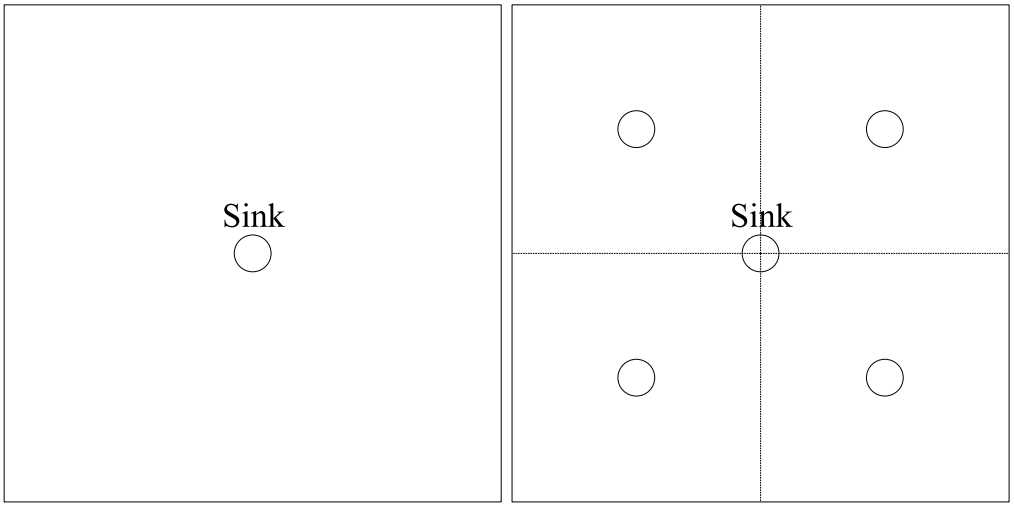
Table 4.1: Summary of the symbols in TED

All the constants are summarized in Table 4.1. $cost_{all}$ may be obtained by nonlinear programming techniques such as generalized gradient search algorithm. In addition to square deployment, other deployment method may also be used and the only difference lies in Equation 4.3, 4.4 and 4.5.

4.4.2 Hierarchical Deployment

Apart from even deployment, the fusion points may also be deployed in a hierarchical fashion. Formally, we assume a square network deployment where the fusion points are distributed in a hierarchical way in the network. We assume that the sink node is at the center of the network. And more fusion points can be added by dividing the network evenly into the square sub-regions of same size. For example, a network with only sink node is shown in Figure 4.3a. If we add 4 fusion points, the fusion points will be evenly distributed in the network as in Figure 4.3b.

We can achieve flexibility with this distribution model since we can add more fusion nodes into the network if we want to decrease the cost for de-



(a) Level 1 fusion point (b) Level 2 fusion points

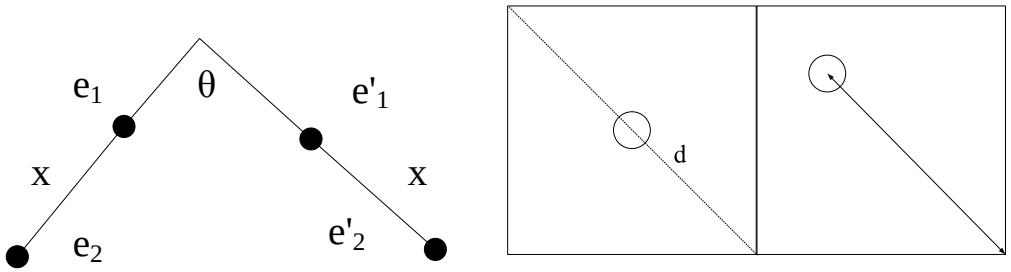
Figure 4.3: Fusion point distribution model in TED

tecting composite events.

For fusion level at ith level, the whole network is divided into 2^{i-1} subregions $region_i$. For example, if the fusion level is at 1 as shown in Figure 4.3a, the region covers the entire network. In Figure 4.3b, the network is divided into 4 subregions at fusion level 2.

In our analytical model, we assume dynamic events where for each event type e_n, it may occur anywhere in the network. In order to include such kind of properties in the model, we define for each $e_n \in E$, it will be monitored by t rounds. The interval between the rounds are T_n. During this $t \times T_n$ time, the set of detected events of type e_n is defined as $E_n = \{e_n^1, e_n^2, \cdots e_n^t\}$.

We measure the event detection cost in terms of message cost. For simplicity, we measure the message cost by the distance (which will in term be reflected as the number of hops) a node can reach another. We uses shortest path tree (SPT) data aggregation protocol [55] as reference to see under



(a) Even distribution (b) Uneven distribution

Figure 4.4: Fusion point distribution

what conditions TED can outperform data aggregation protocols which do consider event relations.

When the fusion points are hierarchically distributed in the network, the composite event detection cost will be bounded by the level of fusion points distributed in the network. Consider Figure 4.4 as an example, if we have a sensor network in a rectangular region where it can be further be divided into two smaller square region. If we assume for each square region, the diameter length is d, then in Figure 4.4a, if we evenly distribute a fusion point in each square region, the max distance for any sensor node in the whole network to reach it's closest fusion point is $\frac{d}{2}$. If the fusion points aren't evenly distributed as shown in Figure 4.4b, then for some nodes in the network, it might have a distance larger than $\frac{d}{2}$ to reach the fusion point.

Theorem 2. *Let the network diameter be d, and the fusion point level be l, the distance between two sub-events e_1 and e_2 is r. After e_1 and e_2 have already been detected, the upper bound of the energy cost for detecting the*

composite event e_3 based on e_1 and e_2 is:

$$cost(e_3) = \frac{d}{2^{l-1}} + r$$

Proof. As shown in Figure 4.4a, the upper bound for one single event to reach a fusion point is:

$$\frac{d}{2^l}$$

Then, the upper bound on the cost for the the two fusion points to reach each other is:

$$\frac{d}{2^{l-1}} + r$$

□

Theorem 2 indicates that if we have no prior knowledge about the events to happen in the network, then fusion points with even distribution can still allow us to have an upper bound on the composite event detection cost.

Theorem 3. *If a node is a level l fusion point, then its distance from sink $r \in D_l$*

$$D_l = \{a_1, a_2, \cdots a_{2^{l-2}} | a_i = \frac{d}{2^{l-1}} \times (2i-1), i < 2^{l-2}\}$$

Proof. When $l = 1$, only sink node is the fusion point. Hence $D_1 = \{0\}$. Similarly, when $l = 2$, $D_2 = \{\frac{d}{2}\}$. We denote each $a_i \in D_k$ as a_i^k. Based on our fusion point distribution scheme, at fusion level $k+1$, we simply divide the sub-regions at fusion level k into four. Therefore, for each $a_i^k \in D_k$, we

can find two elements $a_{2i-1}^{k+1}, a_{2i}^{k+1} \in D_{k+1}$ such that:

$$\begin{cases} a_{2i-1}^{k+1} = a_i^k - \frac{d}{2^k} \\ a_{2i}^{k+1} = a_i^k + \frac{d}{2^k} \end{cases}$$

$$\Rightarrow \begin{cases} a_{2i-1}^{k+1} = \frac{d}{2^{k-1}} \times (2i-1) - \frac{d}{2^k} \\ a_{2i}^{k+1} = \frac{d}{2^{k-1}} \times (2i-1) + \frac{d}{2^k} \end{cases}$$

$$\Rightarrow \begin{cases} a_{2i-1}^{k+1} = \frac{d}{2^k} \times (2(2i-1)+1) \\ a_{2i}^{k+1} = \frac{d}{2^k} \times (2(2i)+1) \end{cases}$$

Hence, Theorem 3 holds for any natural number. □

Theorem 3 may be used as one of the conditions for selecting fusion points.

Chapter 5

Clustering for PSWare

In the previous chapter, we outlined TED, a distributed event detection algorithm. In addition, we described a hierarchical fusion point placement problem. However, these algorithms use generic event probabilistic model and may not perform well compared with more application specific algorithms. Fortunately, PSWare has the flexibility of incorporating different algorithms. In this chapter, motivated by SHM application we describe a novel clustering algorithm that can help to detect events in this application domain.

5.1 Overview of WSN-based SHM

One potential application of WSNs that is far less investigated by computer science researchers is structural health monitoring (SHM). The objective of SHM is to monitor the integrity of structures such as buildings, dams, bridges, and to detect and pinpoint the locations of any possible damage. Unlike

other monitoring applications, detection of possible structure damage is not straightforward and requires significant amount of domain knowledge such as finite element model updating and damage indicator extraction [26]. Moreover, many assumptions which were used to model the network associated problems, such as unit disk or convex sensing region, 0/1 local decision, or data aggregation by average, are unfortunately not realistic in SHM. The required domain knowledge, along with the complexity of SHM, prohibits computer science researchers from investigating this application as intensely as others.

As a result, most research work so far in WSN-based SHM was done by researchers in civil engineering. Using their knowledge in structure engineering, they have designed and developed many WSN-based SHM systems [77][87]. However, based on our experience obtained from the previous collaborations with civil researchers, they generally concern whether the developed WSN-based SHM system can replicate the data delivery functionality of original wire-based counterpart and have less interest to embed in-network processing technology. Moreover, although they have solved many practical engineering problems, they still have difficulties to handle limitations of WSNs such as limited wireless bandwidth, limited communication range, and limited resources of wireless sensor nodes, etc. When designing WSN-based SHM systems, civil engineers sometimes choose powerful wireless sensor nodes to accomplish work that could have been achieved by more cost-effective counterpart through system optimization. This leaves a large space to explore for computer science researchers.

We demonstrate that computer science researchers can help fill this gap

and significantly improve the performance of a WSN-based SHM system. We consider a fundamental problem in SHM: modal analysis, by which the vibrational characteristics of a structure are obtained. These characteristics, called modal parameters, are basis for most of SHM algorithms and can also be used for vibration control and safety assessment. Traditional modal analysis is centralized which needs to stream all the measurement data back to a central unit. This method generally has high energy consumption and low scalability. We describe a cluster-based modal analysis approach. The basic idea of this approach is similar like the 'divide and conquer', where sensor nodes deployed on a structure are partitioned into clusters and modal analysis is carried out in each cluster. The resultant modal parameters of each cluster are then assembled together to obtain the modal parameters of the whole structure. In this approach, clustering is of great importance and should meet some extra requirements of modal analysis. Moreover, cluster size should be optimized to minimize the total energy consumption.

The outline for the rest sections in this chapter is as follows:

1. We show that design of a WSN-based SHM system is a multi-disciplinary area that the efforts from researchers in computer science engineering can significantly help improve system's usability and efficiency.

2. We proposed a cluster-based modal analysis strategy. The clustering problem in this strategy is formulated and proven to be NP-complete. Two centralized and one distributed algorithms are proposed.

$\mathbf{\Psi_k}$	The k^{th} mode shape vector of the structure
p	The number of mode shape vectors to be identified
$G_{xy}(\omega)$	The cross spectral density ($x \neq y$) and power spectral density ($x = y$)
N	the total data amount
M, c, n_i	the total number of sensor nodes, the number of generated clusters, and the number of sensor nodes in cluster S_i
n_t	Length of each section to calculate CSD
n_d	Number of averages
e_S, e_R, e_T	Energy consumed for sampling/rece./trans. one data
e_{NExT}, e_{ERA}	Energy consumed for NExT and ERA

Table 5.1: Summary of Notations

5.2 Structural Mode Shapes

In this section, we will describe the cluster-based modal analysis approach with focus on the optimization of clustering. Before we formulate this clustering problem, the basic concept of modal parameters, the techniques adopted for modal analysis and assembling method are described. Table 5.1 summarizes the notations to be used.

In particular, we give a brief introduction of one important type of modal parameters: mode shapes.

Each mechanical structure has a number of specific vibration patterns at specific frequencies. These vibration patterns are called mode shapes. For example, we deploy a total of m sensor nodes on a structure and extract a total of p mode shapes from the measurement of these sensors:

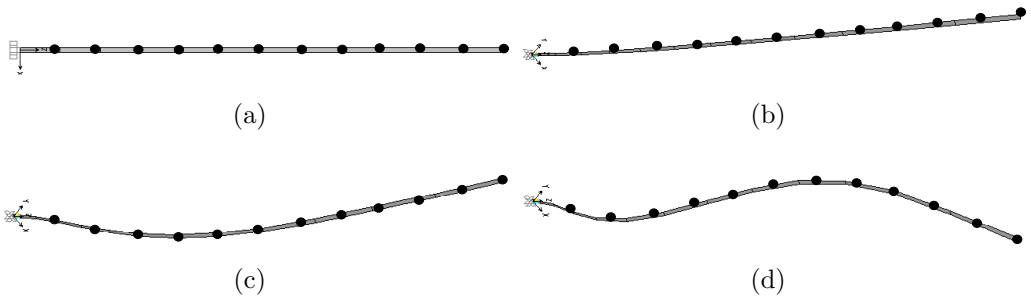


Figure 5.1: Mode shapes of a typical cantilevered beam

$$[\mathbf{\Psi_1}, \mathbf{\Psi_2}, \cdots, \mathbf{\Psi_p}] = \begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1p} \\ \phi_{21} & \phi_{22} & \cdots & \phi_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \cdots & \phi_{mp} \end{bmatrix} \tag{5.1}$$

where mode shape $\mathbf{\Psi_k} = [\phi_{1k}, \phi_{2k}, \cdots, \phi_{mk}]'$ is the k^{th} vibration pattern of the structure. $\phi_{ik}(i = 1, 2, \cdots, m)$ is the k^{th} mode shape value defined at the i^{th} sensor.

As an example, Figure 5.1 illustrates the first three mode shapes of a typical cantilevered beam, extracted from the measurements of the deployed 12 sensor nodes. Figure 5.1a represents the Original beam. Figure 5.1b, 5.1c and 5.1d represents Mode Shape 1, Mode Shape 2 and Mode Shape 3 respectively.

It can be seen that mode shape vector $\mathbf{\Psi_k}$ has an element corresponding to each sensor node. The more number of sensor nodes used, the more elements are contained in $\mathbf{\Psi_k}$, and more accurately this vibration pattern of the structure is described. Considering example in Figure 5.1, if we double the number of nodes deployed on the beam, the vibration patterns will

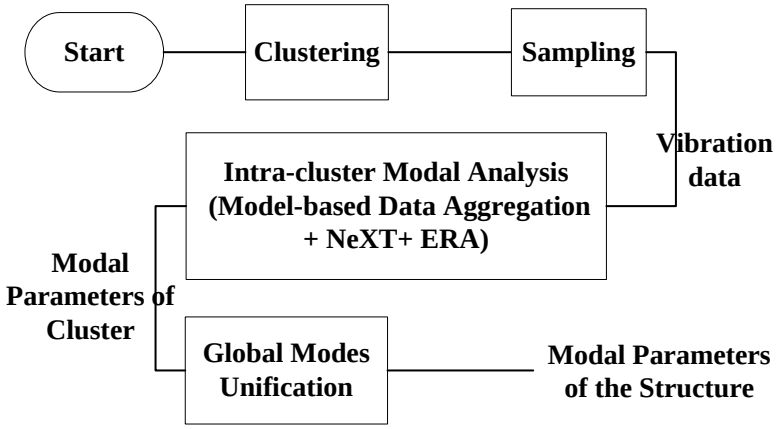


Figure 5.2: Overview of cluster-based modal analysis process

be represented with higher granularity. Another important characteristic of mode shape is that elements in $\mathbf{\Psi_k}$ only represent the relative vibration amplitudes of structure at corresponding sensor nodes. That is, $\mathbf{\Psi_k} = \zeta\mathbf{\Psi_k}$, where ζ is any non-zero real number. This property will be re-visited when we formulate the clustering problem in section 5.3.1.

5.2.1 Clustering for Modal Analysis

In this sub-section, we first give an overview of this cluster-based approach and then formulate the energy consumption of cluster-based modal analysis.

In the cluster-based modal analysis, deployed sensor nodes are partitioned into a number of single-hop clusters and each CH performs intra-cluster modal analysis to extract local mode shapes. Since mode shapes of a cluster only contain elements corresponding to the sensor nodes in that cluster, the mode shapes in all the clusters need to be assembled to obtain the mode shapes defined on all the deployed sensor nodes. The whole process is illustrated in Figure 5.2.

The modal parameters are identified using the natural excitation technique (NExT)[48] in conjunction with the ERA. NexT+ERA is a widely accepted modal analysis approach and can give accurate mode shape estimate using output data-only.

In each cluster, the NExT is used first to calculate power spectral density (PSD) of the CH and cross spectral density (CSD) between the CH and each of the cluster member. PSD and CSD functions are estimated using:

$$G_{xy}(\omega) = \frac{1}{n_d \cdot n_t} \sum_{i=1}^{n_d} X_i^*(\omega) \cdot Y_i(\omega) \tag{5.2}$$

where $G_{xy}(\omega)$ is the CSD between two vibration signals, $x(t)$ and $y(t)$, measured from CH and a cluster member, respectively. $X(\omega)$ and $Y(\omega)$ are the Fourier transforms of $x(t)$ and $y(t)$, and '*' denotes the complex conjugate. n_t is time length of each record $x_i(t)$ or $y_i(t)$. n_d is the number of averages mainly for denoising purpose and n_d practically ranges from 10 to 20. When calculating G_{xy}, consecutive records of $x_i(t)$(also $y_i(t)$) generally overlap. When $y(t)$ in Eq. 5.2 is replace by $x(t)$, the power spectral density (PSD) of CH is obtained.

After obtaining CSD and PSD functions, the inverse Fourier transform is implemented and the cross-correlation functions (CCFs) and auto-correlation function (ACF) are obtained. The ERA uses these functions to build a state space system whereby mode shapes of the structure are identified.

Traditionally, CH collects the raw data from all its cluster members, calculates CSDs and its PSD, and then uses the ERA to identify mode shapes. However, the model-based data aggregation method proposed by [87] can be

used here to decrease the energy consumption. In this approach, instead of collecting measurements data from cluster members, CH broadcasts its time record of length n_t. On receiving the record, each cluster member calculates its CSD and stores it locally. This procedure will be repeated n_d times, until the CSD is according to Eq. 5.2. Each cluster member then transmits the first half of the corresponding CCF to the CH.

Based on the discussion above, we can estimate the energy consumption of intra-cluster modal analysis. To obtain the mode shapes of a cluster S_i, the total energy consumption in S_i, denoted as $cost(S_i)$, can be mainly decomposed into the following three parts:

$$cost(S_i) = Er_s(S_i) + Er_c(S_i) + Er_a(S_i) \tag{5.3a}$$

where $Er_s(S_i)$, $Er_c(S_i)$ and $Er_a(S_i)$ are the energy consumed in data sampling, intra-cluster wireless communication and computation associated with modal analysis, respectively.

Assume a cluster S_i contains a total of n_i sensor nodes, then sampling cost $Er_s(S_i)$ is:

$$Er_s(S_i) = n_i \cdot N \cdot e_S \tag{5.3b}$$

where N is the total amount of time history record sampled in each sensor. Assuming 50 % overlapping, $N = (n_d/2 + 1/2)n_t$. e_S is the energy for sampling one data. We assume that n_d , n_t, N and e_S are fixed.

The intra-cluster wireless communication cost $Er_c(S_i)$ is:

$$\begin{aligned} Er_c(S_i) =& N \cdot e_T + (n_i - 1)N \cdot e_R \\ &+ (n_i - 1)\frac{n_t}{2}(e_T + e_R) \end{aligned} \tag{5.3c}$$

where e_T and e_R are the energy cost for transmitting and receiving one data, respectively. The first two terms at the right side of Eq. 5.3c are the energy consumed when CH broadcasts its time history data and when all the cluster members receive the broadcasts, respectively. The last term is the energy consumption when the $(n_i - 1)$ cluster members transmit back their correlation functions to the CH.

The computation cost $Er_a(S_i)$ can be formulated as:

$$Er_a(S_i) = n_i \cdot e_{NExT} + e_{ERA}(n_i) \tag{5.3d}$$

where e_{NExT} is the energy consumed when each node implements the NExT (including calculating the CSD/PSD and CCF/ACF) and e_{ERA} is the energy used in CH when it carries out the ERA for mode shape identification. e_{NExT} is fixed given n_t and n_d. e_{ERA} is dependent on n_i and number of mode shape vectors p to be identified. Given p, $e_{ERA}(n_i)$ is not a linear function of n_i since the ERA involves complex matrix computations including SVD and matrix inversion. This point is demonstrated in Figure 5.3, where the computation time of our SHM mote to implement the ERA for different cluster sizes is illustrated. The fitting function is also illustrated in the figure. It can be seen that with the increase of n_i, the time consumed, which is the

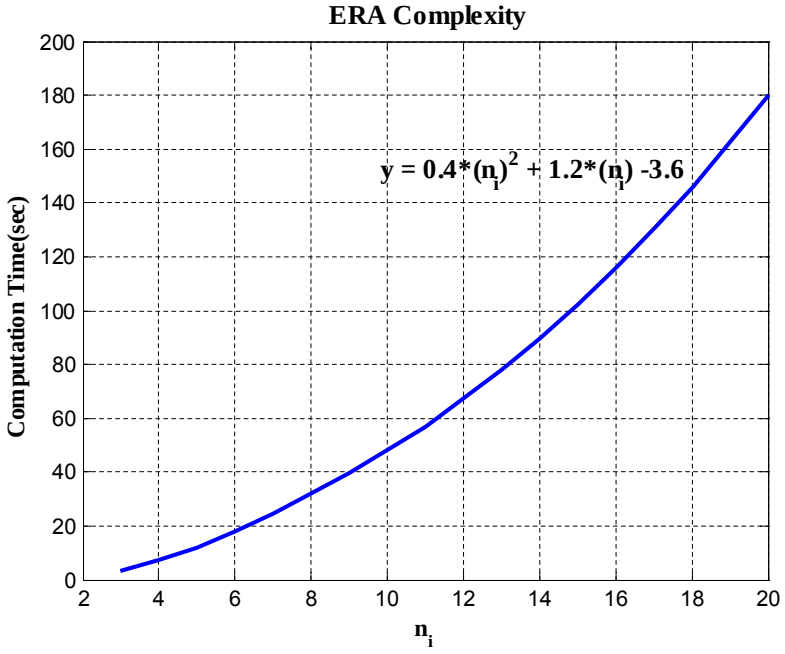


Figure 5.3: The complexity of the ERA

indicator of energy consumption, is quadratically increased.

From the equations above, we have $cost(S_i) = cost(n_i)$, indicating that the energy consumption of a cluster is only associated with the number of sensor nodes in this cluster. It is of interest to see that if possible, whether to generate small-sized clusters or large-sized clusters is more energy efficient. To find the answer, we assume M sensor nodes can be partitioned into equal-sized clusters of size n, then the number of clusters $c = M/n$. The optimal cluster size, denoted as n_{opt}, can be obtained by looking for the n that minimizes the average energy consumption per node defined as:

$$Epn(n) = \frac{c \cdot cost(n)}{M} = N(e_S + \beta) + e_{NExT} + \frac{N(e_T - \beta)}{n} + \frac{e_{ERA}(n)}{n} \tag{5.4}$$

where $\beta = e_R + \frac{n_t}{2N}(e_T + e_R)$.

The 3^{rd} term in the right side of the Eq. 5.4 indicates that in terms of wireless communication, partitioning sensor network into large-sized clusters is preferred when $e_T \geq \beta$ while generating small-sized clusters is better if otherwise. The 4^{th} term tells us that small cluster size n is more energy efficient in terms of computation considering that $e_{ERA}(n)$ is a quadratic function of n. As a result, there does not exist a rule of thumb for clustering and we have different optimal cluster sizes for different conditions. As an example, some parameters obtained by some real tests of our SHM Mote are listed in Table 5.2. Based on Table 5.2, Figure 5.4a shows various optimal cluster sizes, illustrated as red dots in the figure, when the transmission power e_T is set to be from $e_T = e_R$ to $e_T = 5e_R$. It can be seen that when $e_T = e_R$, the smaller the cluster size, the better. (Note that the ERA requires that the number of sensor nodes in each cluster should be at least larger than p). With the increase of e_T, the optimal cluster size is increased but does not go unbounded considering the energy consumption of the ERA for large-sized clusters.

Clustering using minimum dominating set [104] or maximum independent set [6] cannot be directly applied to solve our clustering problem since they mainly aim to find as small number of clusters as possible. Also, in the

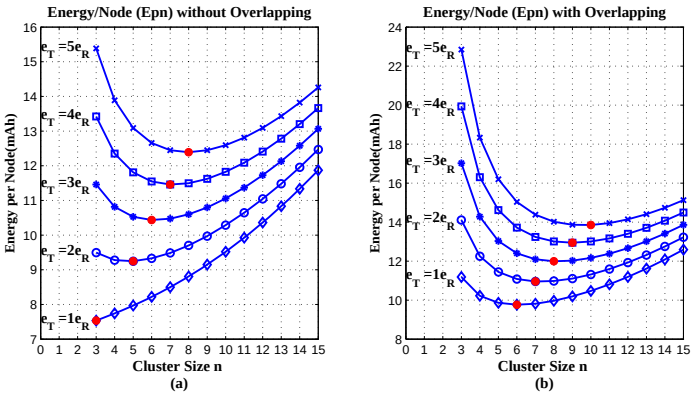


Figure 5.4: The optimal cluster sizes in different conditions.

N	p	n_t	n_d	e_S (mAh)	e_R (mAh)	e_{NExT} (mAh)	e_{ERA} (mAh)
10752	3	1024	20	1.1e-4	5e-4	0.5	$0.0417(0.4n_i^3$ $+1.2n_i$ -3.6)

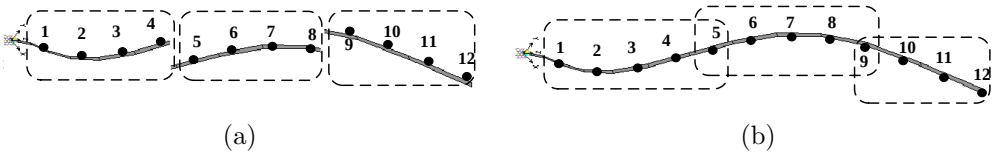
Table 5.2: Parameters used in Figure 5.4

discussion so far, we assume that no overlapping nodes exist in the clusters. However, we will show in the following section that a necessary condition for cluster-based modal analysis is that all the generated clusters must be connected through the overlapping nodes. This requirement further increases the difficulty of the clustering problem.

5.2.2 Mode Shape Assembling

After the mode shapes in all clusters have been identified, they need to be stitched together to obtain mode shapes defined on all of deployed sensor nodes.

However, since mode shape vectors identified in a cluster only represent the relative vibration amplitudes at cluster sensor nodes, mode shapes of



(a) (b)

Figure 5.5: Mode shape assembling

different clusters may not be able to be assembled together. This can be demonstrated in Figure 5.5a, where the deployed 12 sensor nodes in Figure 5.1 are partitioned into three clusters to identify the 3^{rd} mode shape. Although the mode shape of each cluster is correctly identified, we still cannot obtain the mode shapes for the whole structure. The key to solve this problem is overlapping. We must ensure that each cluster has at least one node which also belongs to another cluster and all the clusters are connected through the overlapping nodes (a more formal definition will be given in the next section). For example, in Figure 5.5b, mode shapes identified in each of the three clusters can be assembled together with the help of the overlapping nodes 5 and 9. This requirement of overlapping must be satisfied when formulating the problem of optimal clustering.

It is obvious that overlapping will affect the overall energy consumption and consequently, the optimal cluster size n_{opt} will be different from that when no overlapping is considered. By defining the number of overlapping nodes as $n_o = \sum_{i=1}^{c} |S_i| - M$, and still assume these M sensor nodes are partitioned into equal-sized clusters of size n, then the energy consumption per node becomes

$$Epn'(n) = \frac{(M + n_o)/n \cdot cost(n) - n_o \cdot N \cdot e_S}{M} = \frac{cost(n)}{n} + \frac{n_o}{M} \cdot \kappa \tag{5.5}$$

where $\kappa = N \cdot \beta + e_{NExT} + \frac{N(e_T - \beta)}{n} + \frac{e_{ERA}(n)}{n}$. Considering the fact that unnecessary overlapping will cause extra energy consumption and the number of overlapping nodes should be kept as small as possible, we require that $n_o \geq \frac{M+n_o}{n} - 1$. Therefore,

$$Epn'(n) \geq \frac{cost(n)}{n} + \frac{1 - n/M}{n-1}\kappa \tag{5.6}$$

The right side of Eq. 5.6 essentially provides a lower bound of energy consumption that clustering can achieve when the overlapping constraint is considered. The optimal cluster size n_{opt} can be calculated by minimizing n in Eq. 5.6. For example, the n_{opt} for the parameters listed in Table 5.2 are illustrated in Figure 5.4b.

By comparing Figure 5.4a with Figure 5.4b, it also can be easily seen that optimal cluster size is larger when overlapping constraint is considered. Clustering which generates small-sized clusters may not be energy efficient since a large number of overlapping nodes can cause extra energy consumption in terms of communication and computation.

Also should be noted is that the optimal cluster size n_{opt}, either obtained by Eq. 5.4 or by Eq. 5.6, is not affected by actual network topology. In a dense network, it is more possible to achieve the obtained optimal cluster

size and therefore, the total energy will be lower than a sparse network.

Here, we do not consider the inter-cluster communication simply because delivering obtained mode shapes requires significantly less energy than other processes.

5.3 Clustering Algorithms

In this section, we will formulate the optimal clustering problem and, then, describe the corresponding solutions to the problem.

5.3.1 Problem Formulation

The objective of clustering is that the generated clusters can minimize the energy consumption of overall modal analysis. Clustering also has to satisfy the following constraints (1) each sensor node belongs to at least one of the generated clusters, (2) sensor nodes in each cluster is within a single communication range to its CH, (3) number of sensor nodes in each cluster is larger than p (p:the number of mode shape vectors to be identified) (4) all the clusters are connected together through the overlapping nodes. More formally, problem is formulated as follows: Given a sensor network $G = (V, E)$, find a clustering scheme that can cluster these V sensor nodes into a set of clusters, denoted as $C = \{S_1, S_2, S_3, \cdots\}$, subject to the following constraints:

1. $\bigcup_{S_i \in C} = V$
2. Let the sub-graph for S_i is $G(S_i, E_i)$, where $E_i \subseteq E$. Then $\forall S_i \in$

$C, \exists s_i \in S_i$, such that there is an edge $a_{ij} \in E_i$ between s_i and any other $s_j \in S_i (s_i \neq s_j)$

3. $\forall S_i \in C, |S_i| \geq p$

4. $\forall S_i, \exists S_j \in C, (i \neq j), S_i \bigcap S_j \neq \emptyset$

5. $\forall C' \subseteq C, (\bigcup_{S_i \in C'} S_i) \bigcap (\bigcup_{S_j \in C - C'} S_j) \neq \emptyset$

Objective:

- Minimize $\sum_{S_i \in C} cost(S_i)$

The first constraint is set because we wish to find the mode shapes defined on all the deployed sensor nodes. The second constraint is to ensure only single-hop clusters are generated. Constraint 3 is required by the ERA algorithm. Constraints 4 and 5 are used to describe that generated clusters are overlapping and connected.

The above clustering problem is an optimization problem. We will prove that the decision version of the problem is NP complete which is defined as: *given a threshold k , does there exist a cluster set $C = \{S_1, S_2, S_3, \cdots\}$, which satisfy all the constraints above and whose total energy cost $\sum_{S_i \in C} cost(S_i)$ is equal or smaller than k?.*

Theorem 4. *The decision version of our clustering problem is NP-complete.*

Proof. It is easy to find out this clustering problem is NP. Given a cluster set C, all the constraints above, include constraints 4 and 5 can be checked in a polynomial time. The detailed proof of this part is omitted for brevity.

We show this decision version of the clustering problem is NP-hard by reducing the set cover problem to it. The set cover problem is defined as follows.

Given:

1. A universe V'
2. A set of $S' = \{S'_1, S'_2, ...\} \subseteq V'$
3. The cost function for each subset $S'_i \in S'$: $cost'(S'_i)$
4. A number k'

Find: If there is a subset $C' \subseteq S'$ which satisfies

1. $\bigcup_{S'_i \in C'} S'_i = V'$
2. $\sum_{S'_i \in C'} cost'(S'_i) \leq k'$

To reduce the set cover problem to the clustering problem, we construct a sensor network $G = (V, E)$ from the inputs of set cover problem in the following way: The vertices $V = V' \bigcup X$, where $X = \{x_1, x_2, ...x_p\}$ is a set of p virtual nodes. To construct the edges E, for each $S'_i \in S'$, we first choose an arbitrary node $s'_i \in S'_i$, then we add an edge between s'_i and any other node $s'_j \in S'_i (s'_j \neq s'_i)$. We also add an edge between s'_i and any virtual node in X. The cost function in the clustering problem $cost(\cdot) = cost'(\cdot)$. We also define that by adding/deleting any virtual node x to/from any group will not affect the cost function. The energy threshold $k = k'$.

With this transformation, it can be easily proved that 1) Assume $C' = \{S'_1, S'_2, ...\}$is a solution to the set cover problem, then $C = \{S_1, S_2, ...\}$ is

a solution to the clustering problem, where $S_i = S_i' \bigcup X$. 2) Assume $G = (V, E)$ is constructed from the set cover problem and we have a solution $C = \{S_1, S_2, ...\}$ to the clustering problem, then $C' = \{S_1', S_2', ...\}$ is a solution to the set cover problem, where $S_i' = S_i - X$. The detailed proof is omitted for brevity. □

By reducing the NP-complete set problem to our clustering problem, we have demonstrated that the decision version of our problem is NP-complete. Obviously, the original clustering problem is also NP-Complete.

5.3.2 Centralized Algorithms

Two centralized algorithms are proposed to solve our clustering problem. These two algorithms use the similar idea of the greedy algorithm for the set cover problem but adopt different approaches to handle the extra constraints of clustering. In both of the algorithms, a set of candidate single-hop clusters is first established given the network $G = (V, E)$. Then the most cost-effective cluster is selected from this set, one at a time, until all the sensor nodes in V have been covered.

In the first algorithm, to find a candidate cluster set U, we first calculate the optimal cluster size n_{opt} according to Eq. 5.6. Then based on n_{opt}, one-hop neighbors of each node in V are partitioned. For each node $s_i \in V$, assume the one-hop neighbor set is Ne_{s_i}, if $|Ne_{s_i}| \geq n_{opt} - 1$, then each cluster in the cluster set contains a common element s_i and the remaining elements are the combinations of nodes in Ne_{s_i} with the length of $n_{opt} - 1$. When $|Ne_{s_i}| < n_{opt} - 1$, $C_i = \{s_i\} \bigcup Ne_{s_i}$. Note that we assume the network

is dense enough such that each sensor node has at least p one-hop neighbors. The obtained cluster sets for all the nodes in V are combined together to obtain the candidate cluster set U.

The algorithm then selects the most cost-effective cluster $S_i \in U$, one at a time, until all the sensor nodes in V have been covered. The cost effectiveness, denoted as λ, is defined as $\lambda = \frac{1}{|S_i \cup C_a| - |C_a|}$, where C_a represents the set of nodes covered so far. When selecting the most cost-effective cluster, we choose from the clusters in U which overlap with C_a. This strategy can ensure that all the selected sensor nodes will be connected through the overlapping nodes. If more than one candidate clusters which overlap with C_a have the same λ, the one which maximizes the total degrees of the remaining un-covered nodes (i.e. $U - S_i \bigcup C_a$) will be chosen. It can be seen that this algorithm divides the sensor nodes in V into as many single-hop clusters of size n_{opt} as possible while keeps the number of overlapping nodes into minimums (from $\lambda = \frac{1}{|S_i \cup C_a| - |C_a|}$, penalty is given to cluster having large number of overlapping nodes with C_a). Both of these two points are of importance to minimize the overall energy cost. The algorithm is shown as Algorithm 8.

The second algorithm uses different strategy to handle overlapping. First, the optimal cluster size n_{opt} is calculated based on Eq. 5.5 without considering overlapping constraint. When selecting the most cost effective cluster, it is chosen from all the candidate clusters in U. Since the overlapping constraint is not considered when selecting cluster, after all the sensor nodes in V have been covered, the algorithm will test if all the clusters are connected through the overlapping nodes and add extra clusters to connect them if

necessary. The basic idea is to identify all the isolated cluster groups and then find clusters to connect them. The detailed description is omitted for brevity. This algorithm is shown as Algorithm 9.

Algorithm 8 First centralized algorithm for clustering

Input: $G = (V, E)$ and parameters listed in Table. 5.2

find n_{opt} which minimizes Eq. 5.5
$U \leftarrow \emptyset$ $C_a \leftarrow \emptyset$
for all $n_i \in V$ **do**
  $S_i \leftarrow \emptyset$
  for all one hop neighbor n_j of n_i **do**
    $S_i = S_i \bigcup \{n_j\}$
  end for
  construct a cluster set C_i whose elements are the combinations taken of the nodes in S_i of length $n_{opt} - 1$.
  $U = U \bigcup C_i$
end for
repeat
  C_{cand} = all the clusters in U which overlap with C_a
  find a cluster S_i in C_{cand} with the smallest $\frac{1}{|S_i \cup C_a| - |C_a|}$
  $C_a = C_a \bigcup S_i$
until C_a covers V

Output: C_a

5.3.3 Distributed Algorithm

Based on our first centralized algorithm, we propose a distributed solution. In this solution, each node only needs its one-hop neighbors information and communicates only with its one-hop neighbors. The clustering will start at a single controller node which usually is the sink node of the network.

Similar to Algorithm 8, each newly created cluster will be connected to at least one of the existing clusters. In the distributed algorithm, each node will maintain two lists of neighbors: unclustered and clustered. The lists will

Algorithm 9 Second centralized algorithm for clustering

Input: $G = (V, E)$ and parameters listed in Table. 5.2

find n_{opt} which minimizes Eq. 5.4
The same with the 2 to 16 lines of Algorithm 8
repeat
    find a cluster S_i in U with the smallest $\frac{1}{|S_i \cup C_a| - |C_a|}$
    $C_a = C_a \bigcup S_i$
until C_a covers V
Identify Isolated cluster groups (ICGs) in C_a
Construct a graph $G_{ICG} = (V_{ICG}, E_{ICG})$:
Run MST algorithm on G_{ICG} and get T
for all edges in T **do**
    Create an extra cluster C_e and add it to C_a
end for

Output: C_a

be sorted according to each neighbor's own number of unclustered neighbors. The nodes with fewer unclustered neighbors will do clustering or join other clusters first. This is done by assigning each node's execution of the algorithm to a specific time slot.

Each node has only three roles during clustering: unclustered, CM (cluster member) and CH. We illustrate the pseudo code based on the three roles in Algorithm 10.

Each node will not execute the algorithm until the start of its own time slot. The input p is the minimum cluster size constraint and n_{opt} is the calculated optimal cluster size. Once a CH decides to choose certain node as CM, it will send a message req_{ch} and the corresponding CM will send $acpt_{ch}$ to acknowledge the selection. After the execution of the algorithm on a node, an unclustered node will become CH. At the end of the algorithm, each node will merge all its neighbors into one sorted list and assign time

Algorithm 10 Distributed algorithm for clustering

Input: n_{opt}, p, unclustered neighbors un, clustered neighbors cn (un and cn are both sorted in increasing order according to the number of unclustered neighbors)

1: **if** self is unclustered **then**
2: Self becomes CH
3: **end if**
4: **if** self is CH **then**
5: Select one node from cn as its member
6: **if** $size(un) \geq n_{opt}$ **then**
7: Construct the cluster as size of n_{opt} by selecting first n_{opt} nodes from un as CM
8: **else if** $n_{opt} > size(un) \geq p$ **then**
9: Construct a cluster by selecting all nodes in un as CM
10: **else**
11: First construct a cluster by selecting all nodes in un as CM then select more nodes from cn as CM until $cluster\_size = p$
12: **end if**
13: **else if** self is CM **then**
14: Broadcast a message to all neighbors saying the status is currently CM.
15: **end if**
16: $an = merge(un, cn)$
17: **for all** $n \in an$ that haven't been assigned a time slot **do**
18: Assign time slot $t[i]$ to n where i is the index of n in an
19: **end for**

slots. The duration of the time slot is large enough so that a CH can perform the selection. $t[i]$ is the i^{th} time slot from the end of the current time slot. It can be easily seen that the generated clusters from the distributed algorithm can satisfy all the constraints.

Informally, it can be easily shown that the distributed algorithm is correct as the solution it produces will satisfy all the constraints:

- Since all nodes will be given a time slot to run the algorithm, the clusters will eventually cover the whole network.
- The clustering starts at the controller node and all the later CHs will first select a CM from its cn. Therefore, all the clusters will be connected.
- CH only selects CM within its one-hop neighbors, and the CHs will not remove its role as CH so all clusters will be one-hop.
- For each CH, it will ensure the cluster size is at least p by selecting additional CM until the cluster size reaches either p or n_{opt}.

A Thesis Submitted by Steven Lai

Chapter 6

System Implementation

Since the goal of using middleware is to support application development. We have implemented PSWare on Crossbow's MicaZ sensor nodes with TinyOS 2.x. In this chapter we illustrate how PSWare can be implemented for different applications.

6.1 ITS Implementation Using PSWare

We use our middleware to implement a traffic management prototype system in our testbed. The application architecture is shown in Figure 6.1. PSWare acts as the interface between WSN and application. On the application side, we have a web server where application-specific event templates are defined there. In the traffic management, we have defined events such as traffic jam and collision. Then the application users can subscribe the service by making use of the event templates. They can further refine their application requirements by providing parameters to the event templates.

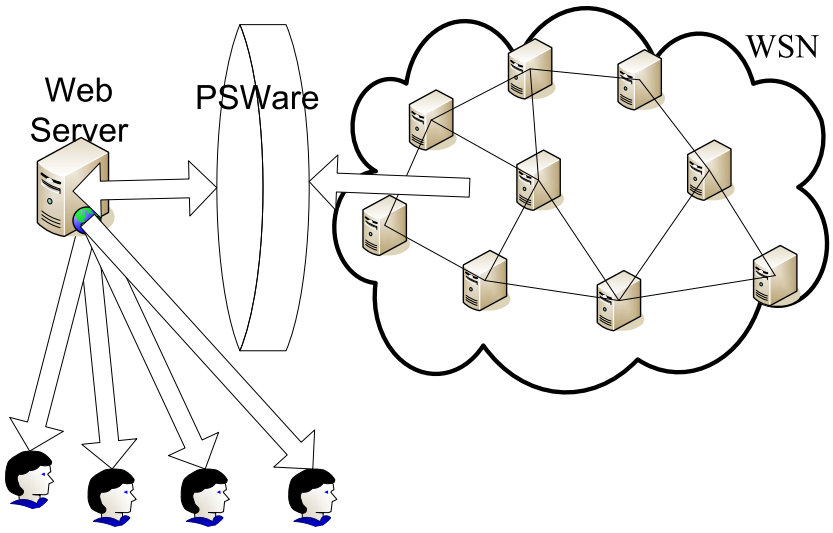


Figure 6.1: Demo application architecture

6.1.1 Pre-defined Events

In order for the users to use our middleware easier, we have provided some pre-defined event templates. With adjustable parameters. Listing 6.1 shows the most basic event - a car event in the system. All the attributes of the event can be directly obtained from the sensor nodes. The user can further add their own defined filters such as selecting only cars from certain road or selecting a car with pre-defined ID.

Listing 6.1: Car event

```
1 Event CarEvent {
2   int roadID=System.roadID;
3   int carID=System.carID;
4   int posID=System.posID;
5   int speed=System.speed;
6 }
```

Based on the basic car event, we can further make use of it and define a traffic jam event as shown in Listing 6.2. In the example, we defined a traffic jam as having more 3 cars on a single road section.

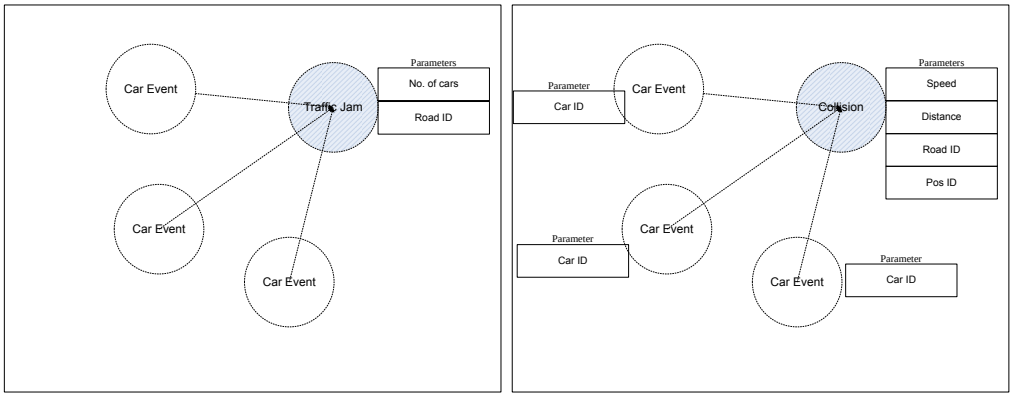
Listing 6.2: Traffic jam event

```
1 Event TrafficJam {
2   int roadID=System.roadID;
3   int carNo=count(c);
4 } on {
5   CarEvent c;
6 } where {
7   roadID=1 &&
8   carNo>3
9 }
```

Another type of event is car collision event. It is defined in terms of two car events. Listing 6.3 defines a collision event on a road if the two car's speed are above certain threshold and their distance are too close.

Listing 6.3: Car collision event

```
1 Event Collision {
2   int roadID1=c1.roadID;
3 } on {
4   CarEvent c1, c2;
5 } where {
6   c1.roadID==c2.roadID &&
7   c1.speed-c2.speed>5 &&
8   c1.posID-c2.posID<2
9 }
```



(a) Traffic jam GUI (b) Collision GUI

Figure 6.2: GUI Design

6.1.2 User Interface Design

We use a graphical user interface (GUI) to visualize the relations between events. Users can also specify their parameters on the given event templates. For example, for a traffic jam event in Figure 6.2a, user can further specify the condition for a traffic jam such as the number of cars waiting on a road. Similarly, for collision events 6.2b, the user can specify the condition for such kind of events. Moreover, the user can also choose to monitor certain cars in the system for the collision event. This can be done by adding a parameter in the car events.

6.1.3 Customized Event Detection for ITS

Intelligent transportation system may cover many application areas including collision avoidance, traffic light control and vehicle tracking. All these applications require the detection of vehicles. In this paper, we consider the scenario where sensor nodes are deployed as road side units (RSU) [54]. To

detect vehicles in such a model, the sensors should be waken up as the vehicles are driving along the roads. The procedure for nodes' wake-up and forwarding is shown in Procedure 11. We use the following notations in the procedure:

- $V = \{v_1, v_2 \cdots\}$: the sensor nodes that come next on the road. For the sensors on the road, $\|V\| = 1$ but for the sensors next to the crossroads, $\|V\| > 1$
- Vehicle events e_i which are either detected locally or received from another node.

Procedure 11 Event forwarding for ITS

```
1: if detected vehicle event e_i then
2:    for all v_n ∈ V do
3:       forward e_i to v_n
4:    end for
5: end if
6: if received vehicle event e_i from v_m then
7:    start data collection
8:    if detected vehicle event e_j then
9:       select e_i and e_j for composite event detection
10:   else
11:      wait for event to expire
12:   end if
13:   stop data collection
14: end if
```

Except for the sentry node, nodes will not actively collect data until they receive messages from others. Once the event from the previous node on the road is received, it will be selected for composite event detection.

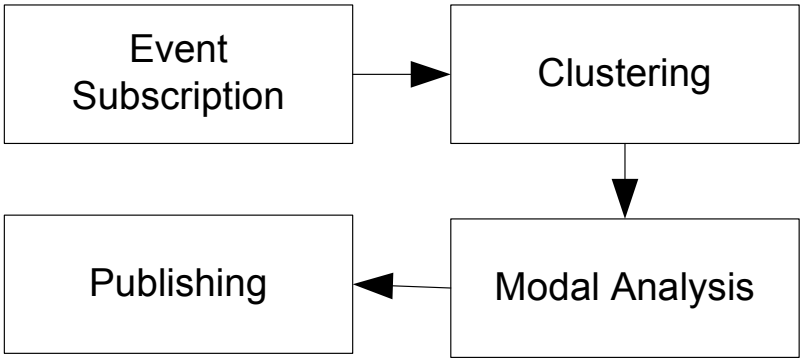


Figure 6.3: SHM Operation Flow

6.2 SHM Implementation Using PSWare

Apart from the ITS applications, we also implement an SHM prototype system. As discussed in the previous chapter, our SHM system involves the following steps:

1. Establish clusters in the network
2. Data sampling and processing to obtain the mode shape
3. Detect possible damage based on the mode shape

When developed using PSWare, the operation flow is shown in Figure 6.3. First, the event subscription will trigger the network to cluster. Then, data will be sampled. During data sampling, we store the sampled data in an internal buffer. Domain experts use modal analysis to obtain the mode shape. Modal analysis is implemented as a special instruction by the domain experts. Once the results are obtained, they will be published to the subscribers.

Since the modal analysis part requires a lot of domain knowledge and is not the focus of this work, in this section, we primarily discuss how the

clustering algorithm can be integrated in PSWare. For our prototype system, we have one sink node and subscriptions are disseminated through the sink node. Such setup is similar to many existing SHM systems where the data are transmitted to a monitoring station for damage detection.

6.2.1 Neighbor Information Exchange

Sensor nodes exchange information periodically. Each sensor node maintains the following information:

- Its own clustering status. There are three possible values: UC (unclustered), CH (cluster head) and CM (cluster member)
- Its neighbor table with clustering status: $table_n$

Each node v_n periodically broadcasts messages msg_n which is its $table_n$. The procedure is shown in Procedure 12.

Procedure 12 $table_n$ update

Input: $v_n \rightarrow msg_n$

for each entry t' in msg_n **do**
  if not exists $t' \rightarrow fid_n$ in $table_n$ **then**
    $addTo(table_n, t')$
  end if
  for each entry t in $table_n$ **do**
    if $t \rightarrow status! = t' \rightarrow status$ **then**
      $t \rightarrow status! = t' \rightarrow status$
    end if
  end for
end for
$msg_n \leftarrow table_r$
$periodically\_broadcast(msg_n)$

6.2.2 Clustering

The actual clustering takes place after the subscription is received by an individual sensor node. Internally, this is done by implementing the 'OPinstall' instruction. This is a special instruction which will be invoked when a sensor node receives new subscriptions. Please refer to Appendix for more information on the instructions.

Procedure 13 Clustering in SHM

Input: OPinstall

```
1: if self is sink then
2:    self.status = CH
3: else
4:    self.status = UC
5: end if
6: while self.status == UC do
7:    broadcast(msg_n)
8:    cost_n = MAX
9:    for each entry t in table_n do
10:      cost'_n = 0
11:      for each entry t' in t.table_n do
12:         if t'.status == UC then
13:            cost'_n = cost'_n + 1
14:         end if
15:      end for
16:      cost_n = min(cost'_n, cost_n)
17:   end for
18:   run_uc();
19: end while
20: if self.status == CM then
21:   run_cm();
22: else
23:   run_ch();
24: end if
```

The procedure is shown in Procedure 13. The procedure starts off from the sink node selecting itself to be a cluster head. The rest of the nodes are at

first unclustered. Then each node will sort its neighbor's list according to the number of unclustered nodes that each neighbor has. This is to determine the neighbors which are more 'isolated' - those with most of its neighbors already clustered. The rest of the part has already been described in the previous chapter. Each node runs run_uc(), run_cm() and run_ch() based on their clustering status. These functions implement the distributed time-slot based clustering algorithm discussed in the previous chapter.

A Thesis Submitted by Steven Lai

Chapter 7

System Evaluation

In this chapter, we analyze the performance of PSWare through analysis, simulation and experiments.

7.1 Analysis on TED

Since TED is the essential algorithm used in our application, our analysis mainly focuses on its energy efficiency and delay. As discussed in our problem formulation, we message cost for measuring the energy efficiency in TED.

7.1.1 Analysis on Message Cost

In order to analyze the efficiency of TED without losing generality, we assume that sensor nodes are randomly deployed in a circular area with radius R. We use distance to approximately measure the number of hops in order to calculate the message cost for event detection. There is no existing work that utilizes event types to detect composite events and it is also very hard

to aggregate events without knowing their actual relations. Therefore, as a reference to compare the message cost, we use shortest path tree (SPT) algorithm where events are collected at the sink because their definitions are not considered for event detection.

We use a similar event model that has been introduced in Section 4.2 for analysis. Moreover, since the actual cost of TED will depend on event probabilities. We include such information in our model as well. Suppose we have two event types e_1 and e_2 which are the two sub-event types for a composite event e_3 (i.e. $e_1 r e_2 = e_3, r \in R$). The probability for e_1 and e_2 to occur is $P(e_1) = p_1$ and $P(e_2) = p_2$ respectively. The probability for e_3 to occur when both e_1 and e_2 have occurred is $P(e_3|e_1, e_2) = p_3$.

In TED, each node periodically broadcasts its routes to the fusion tables so that others can know how to reach the fusion points. Such cost is similar to many existing routing protocols in WSN such as [101] where each node periodically broadcasts its route metrics to the sink for the purpose of link quality evaluation. Therefore, in TED, we mainly consider three types of messages which will be used:

- The overhead for initial event forwarding: $cost_f$
- The overhead for the fusion points to send feedback: $cost_b$
- The message cost for detecting the actual composite events: $cost_d$

Let the average distance between two random nodes in the square region be D and the average distance between a node and sink be d.

The cost for initial event forwarding will be: $cost_f = 2 \times D$. In order to avoid the extra cost for building up the overlay for each fusion point to

communicate with individual sensor nodes, the fusion nodes simply flood the feedback in the network. Therefore, the cost for the fusion points to send feedback will be: $cost_b = D \times (|N| - 2)$. The cost for detecting each individual event is: $cost_d = D$. The total expected message cost using TED over a time period T is:

$$\begin{aligned} & cost_{TED} \\ =& cost_f + cost_b + T(p_1 cost_d + p_2 cost_d + p_3 p_2 p_1 d) \\ =& D|N| + TD(p_1 + p_2) + Tdp_1 p_2 p_3 \end{aligned}$$

Here T is the $expire_n$ used in $table_e$. The total expected message cost using SPT over a time period T is:

$$cost_{SPT} = d \times T(p_1 + p_2)$$

To see when TED will cost less than SPT, we have:

$$\begin{aligned} & cost_{TED} < cost_{SPT} \\ D|N| + TD(p_1 + p_2) + Tdp_1 p_2 p_3 &< dT(p_1 + p_2) \\ T((p_1 + p_2)(d - D) - dp_1 p_2 p_3)) &> |N| \end{aligned}$$

The inequality can be viewed as a trade-off between message cost saved by TED and the overhead. Simply speaking, TED can reduce message cost when:

- Fusion points are closer to event source

- The probability of primitive event is high while the probability of the composite event is lower.

- Each time after the $table_e$ is constructed, it used for a relatively long period of T

7.1.2 Analysis on Delay

In Algorithm 7, if there is no local match, the fusion point will wait for some time to see if there is any other events forwarded from other nodes. In this section, we study the delay of TED.

Theorem 5. *Suppose the sensor nodes are randomly deployed in a circular area where sink is located at the center of the network, the events have a time span of t, fusion points are randomly distributed with an average distance to other nodes of d and the sensor nodes forward the events at the speed of v. Compared with SPT in the worst case, TED will introduce a delay of:*

$$delay_{TED} - delay_{SPT} = \frac{d}{v} + t$$

Proof. Let K be the average distance from an event to the sink. The delay of SPT will be $delay_{SPT} = \frac{K}{v}$. To calculate the delay of the worst case in TED, we first need to calculate when TED will eventually forward the event to the sink. According to Algorithm 7, an event fusion point will forward the event if the sink is closer than any other fusion points. Suppose after k fusion points, the event will be forwarded to the sink. Then, we can get the

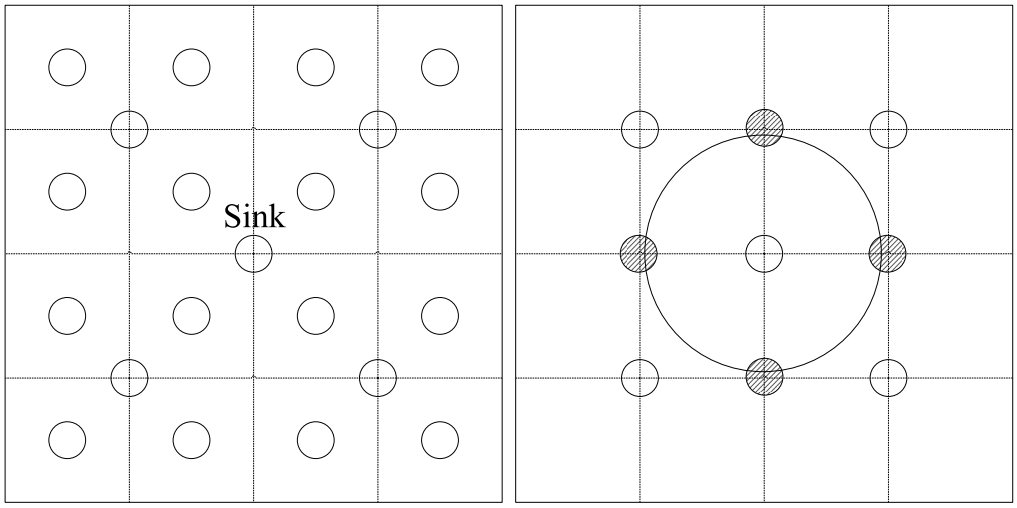
following.

$$
\begin{aligned}
kd &\geq K \\
k &\geq \frac{2}{R}\frac{2}{\pi}\int_0^{\pi} 2cos(\frac{\theta}{2})sin(\theta)[\theta - sin(\theta)]d\theta \\
k &\geq \frac{128R}{45\pi} \times \frac{2}{R} \\
k &\geq 1.81
\end{aligned}
$$

Therefore, we have:

$$
\begin{aligned}
delay_{TED} &= (\frac{d}{v} + t)(k-1) + \frac{D}{v} \\
delay_{TED} - delay_{SPT} &= (\frac{d}{v} + t)(k-1) \\
&= \frac{d}{v} + t
\end{aligned}
$$

□



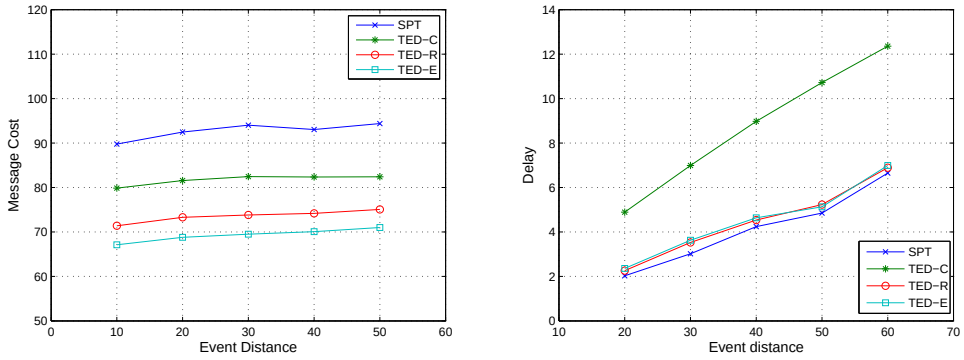
(a) Possible fusion points (b) Communication range

Figure 7.1: Simulation environment

We have conducted both simulation and experiments on PSWare to evaluate its performance. The simulation is used mainly to validate our analytical results on TED while the experiments are done according to some real WSN-based applications in order to demonstrate the effectiveness and efficiency of PSWare.

7.2 Simulation

We use simulation to validate our analytical results. Our simulation is based on TOSSIM [65]. We have 127×127 sensor nodes placed on a square space. We use both even distribution and random distribution of the fusion points. Similar to our analysis, we divide the whole area into small squares and deploy equal number of fusion points for each square. Each sensor node is able to communicate with its nearby neighbors. To simulate the event detection,



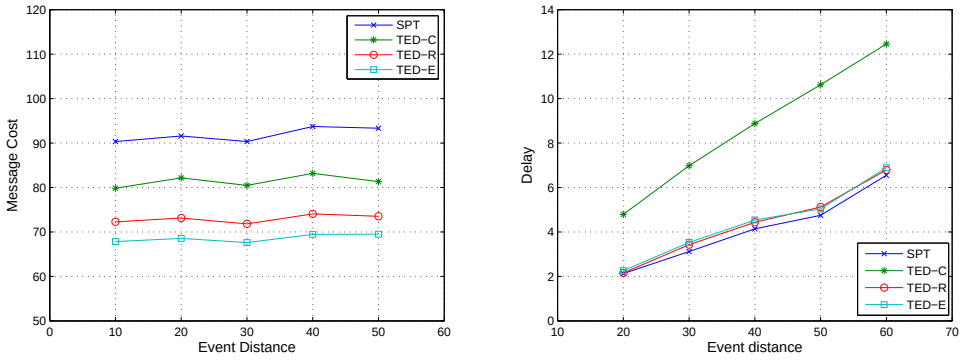
(a) Cost (b) Delay

Figure 7.2: Average event size: 5 nodes

we first randomly generate event sources in the network. Then based on these event sources, we further generate more sub-events that have relations and let the sensor node detect the composite events. We study the performance of TED under different parameters such as event distribution, event probability, and deployment approach. In the figures, TED-R represents distributed TED with random fusion point deployment and TED-E represents distributed TED with even fusion point deployment.

Similar to our analysis, we use SPT as a reference to compare with TED so as to see how much cost we can save if we detect the events using their relations. For simplicity, we assume in SPT approach, the events can always be aggregated at the shared paths. We study the performance of TED under different parameters such as fusion point deployment, event distance, event size and event probability.

To obtain more accurate results, we repeat the simulation for 10,000 times for each of the graph we discuss in the following sections. For each set of data, the standard deviation is less than 0.5. We measure the simulation



(a) Cost (b) Delay

Figure 7.3: Average event size: 10 nodes

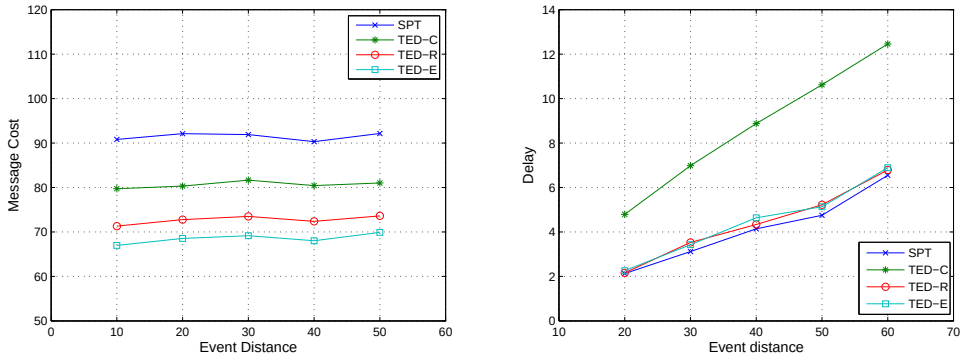
results with three metrics:

- Event distance: the average number of hops between two events
- Event size: the average number of nodes involved in detecting events
- Event probability: the average probability for an event to happen

For each simulation, we fix two parameters to demonstrate the impact of the other parameter to overall message cost.

7.2.1 Impact of Event Distance

The first parameter we study is the distance between events. More precisely, we study the average event distance where it is defined as the average of all the distance between any two events in the network. Figure 7.2 shows the result when average event distance equals to 30. As we can see, SPT introduces more message cost. Among the different versions of TED, TED-C has highest message cost while TED-E has the lowest. In terms of the day,



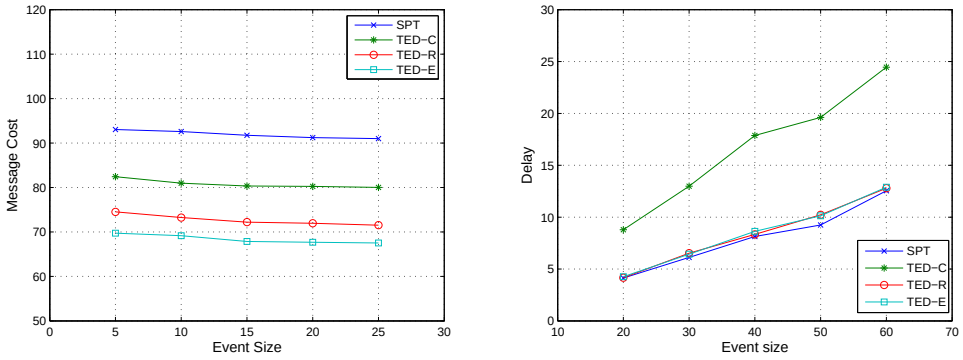
(a) Cost (b) Delay

Figure 7.4: Average event size: 15 nodes

TED-C introduces most delay because the events will have to be collected in a centralized fashion first. The rest of the three have similar delay with SPT slightly lower than the others. Delay increases naturally as the distance increases. Such results agree with our analysis.

The next set of simulation results are shown in Figure 7.3. In this figure, we can see as the event distance becomes greater, the average message cost has also increased accordingly. This is inevitable because the nodes will have relay the events for more number of hops in order to detect the composite events. In terms of the delay, the results are similar to those in the previous set. Note that the difference between this set of simulation and the previous set is the event size. We will get into more details about the impact of event size in the next section.

The last set of results for event distance is shown in Figure 7.4. In this figure, the event size is set to be 15 nodes. Such scenario should cover a lot of WSN applications where the events need to be detected by multiple sensor nodes collaboratively. As we can see from the figure, when the event size



(a) Cost (b) Delay

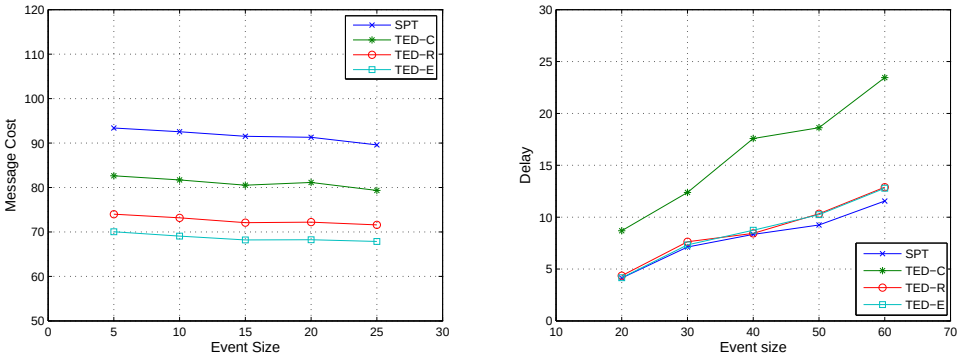
Figure 7.5: Average event distance: 30

becomes larger, the results have also become more stable. This is because as the size becomes larger, the cost between different nodes is amortized. In terms of delay, it's still quite similar to the previous results and event size do not have a significant impact on it.

In summary, as we can see from all the figures, TED can reduce message cost in comparison with SPT. However, among all the simulation parameters as we will see in the following sections, event distance does not seem to be a critical factor where TED can save much cost. In terms of the delay, longer event distance will introduce longer delay to all the algorithms because the nodes simply need to use more hops to reach the sink. In addition, centralized TED introduces longer delay than others because it needs to gather all the events and disseminate the event fusion information to the network.

7.2.2 Impact of Event Size

The next factor we are going to study is event size. Event size is defined as the number of nodes involved in detecting a particular event. For composite



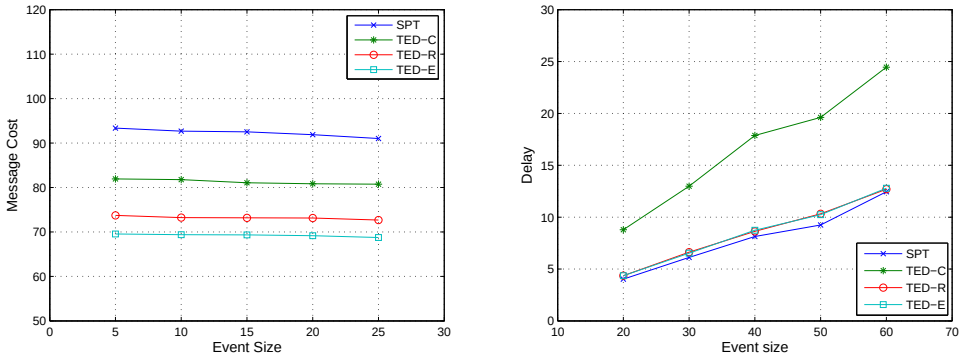
(a) Cost (b) Delay

Figure 7.6: Average event distance: 40

events, it represents all nodes in detecting their sub-events. During composite event detection, sub-events may need to be forwarded to the fusion points for further detection. The detection cost, therefore, also includes the cost for forwarding the events. This factor is interesting to our simulation because different event size may have different effect on the number of nodes that are needed to detect more than one events. It can, therefore, affect the choice of event fusion points since sometimes it may be better to choose the nodes that can detect multiple events as the fusion points.

The first set of results are displayed in Figure 7.5. Similarly, TED has lower message cost than SPT while SPT has smaller delay in comparison with distributed TED.

The next set of results are shown in Figure 7.6. Since based on our previous results, event distance is not a major factor for TED to message cost, we set it to a constant value of 40 in this simulation. We use such large value so that events will have less overlapping. Otherwise, if all events have an overlapping, the fusion point will eventually be selected in the overlapping



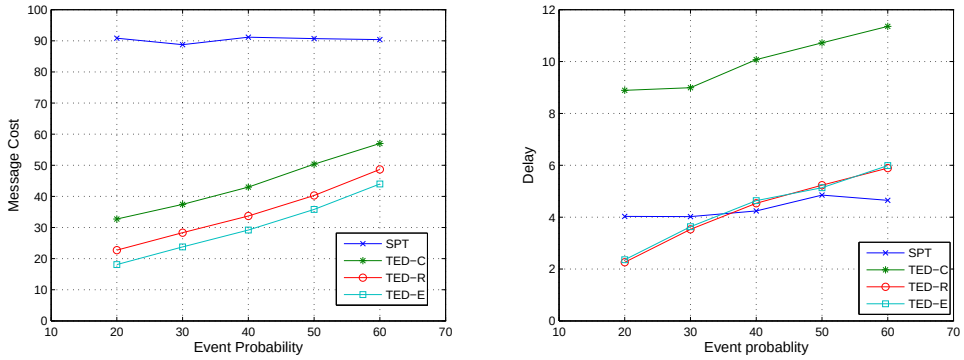
(a) Cost (b) Delay

Figure 7.7: Average event distance: 50

area and this reduces TED to SPT. Since the event size has increased from 5 to 10 nodes, we also need to decide which nodes out of all the event detection nodes will be used as the decision maker for that event. In order to have a reasonably easy function to calculate the cost without losing generality, we choose the node at the geographical center of the event to be the detector node.

As we can see, SPT still incurs the highest cost. Among all the TED variations, the centralized version has relatively higher cost while the distributed TED with even fusion point deployment has the lowest cost. This is because the centralized version will introduce control overhead and the even fusion point deployment strategy guarantees that each event can find a fusion point within certain number of hops. The results for delay is also similar to the previous results so the event distance will not have a significant impact on event delay either.

Our last set of results are shown in Figure 7.7. In this set of results, we set the average event size to be 15 nodes. Note that this is already a

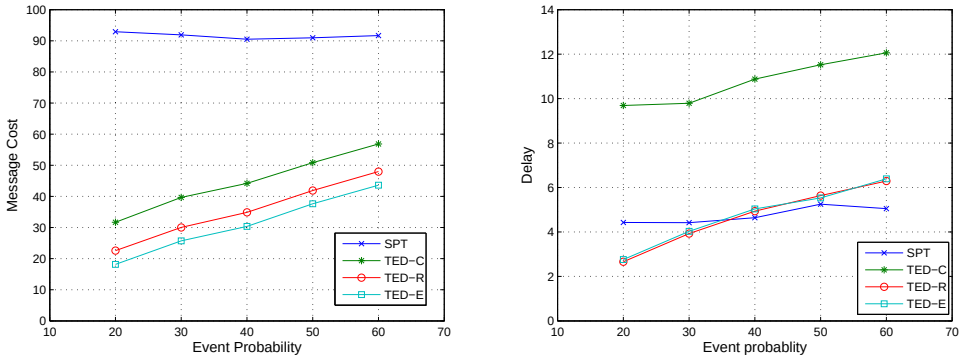


(a) Cost (b) Delay

Figure 7.8: Event size: 5, distance: 10

quite large event in comparison with most primitive events where only one node is used for detection. It may be suitable for many applications where the aggregate functions such as average or sum need to be evaluated. The results are similar to the previous ones in the sense that SPT has the highest message cost while TED-E has the lowest. However, this set of results are more stable and consistent than the previous set. This is because as the event size becomes larger, it is easier to cover more number of nodes and therefore having more overlapped ones. TED may make use of such overlapped nodes to select fusion points. Therefore, TED will have a wider choice as the event size becomes larger and the results will become more stable. The delay, on the other hand, is still quite consistent with the previous results.

In summary, according to all the results shown in this section, we can draw conclusion that event size is not a major factor that will decide the amount of message cost that TED could save. While TED does perform better than SPT in respect of event size and event distance, we still need to study more factor to see if it can perform even better.



(a) Cost (b) Delay

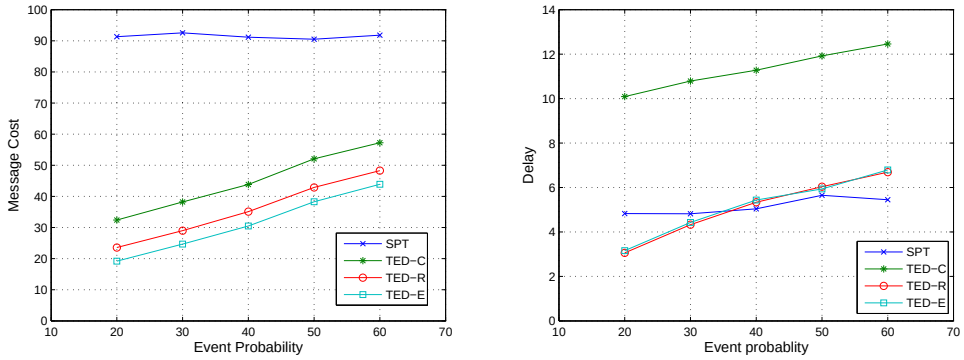
Figure 7.9: Event size: 5, distance: 20

7.2.3 Impact of Event Probability

Since we have studied two parameters in our previous sections, we will study the last factor in accordance to the previous factors we have studied and see how it can affect the performance of TED. Figure 7.8 shows our first set of results. Our first set of results look promising since all variations of TED saves significant amount of message cost than SPT and for some cases, the delay for TED is even lower than that of SPT.

Our second set of results are shown in Figure 7.9. Since we have already concluded in our previous section that event size is not a major factor that will affect the performance of TED, in this set of results, we fix the event size while increasing the event distance for each set. Then within each set, we use the event probability as the X-axis to have a more direct view of the relation between event probability and energy cost.

As shown in the figure, SPT still has the highest message cost. For TED, the message cost is particular low while the event probability is less than



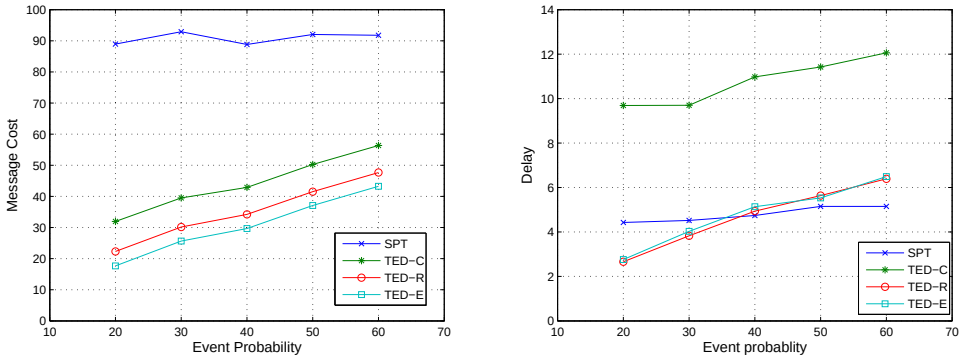
(a) Cost (b) Delay

Figure 7.10: Event size: 5, distance: 30

50%. This is because with TED, we can effectively filter the primitive events if they are not going to lead to higher level of composite events. Such filtering also has an impact on the delay. Unlike SPT where all the events must be forwarded to the sink to decide if a particular composite does not happen, TED can filter out such composite events in the network.

Our third set of results are shown in Figure 7.10. This set of results are consistent with the previous two sets. For SPT, the message cost does not change much with the event probability since all the data will be collected by the sink regardless of the event probability. For all versions of TED, the event detection cost is below 40 when the event probability is 20%. This is because with such a low probability, many of the primitive events have been filtered before they get a chance to be fused for higher level composite event detection.

Among the three variations of TED, we can see TED-C has slightly higher message cost than the other two. This is because of the overhead introduced in the centralized approach where the event fusion points need to be selected



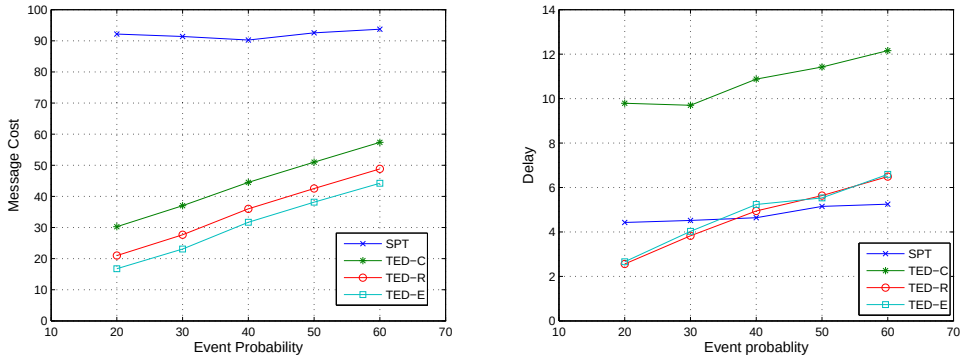
(a) Cost (b) Delay

Figure 7.11: Event distance: 20, size: 10

by the sink node. For the distributed variations, TED-R and TED-E have similar performance. TED-E has slightly lower message cost because of its even distribution. Each sensor nodes may find a fusion point in a bounded distance. In terms of the delay, while centralized TED still has the highest delay, distributed TED will introduce less delay when the events have lower probability and can be filtered in the network.

In our previous results. We studied the impact event probability by fixing event size while increasing the event distance. The fixed event size was set to be 5 which is small enough to be considered as atomic event size. As we have also studied the impact of event size in our previous sections, for the sake of completeness, we also perform another three set of simulations to study the event probability while fixing the event distance.

Since our network size is 127 by 127, we choose the fixed average event distance to be 20. This is around $\frac{1}{6}$ of the network dimension which is probably representative enough for many WSN-based application. Our fourth set of results are presented in 7.11. In the figure, we can also see TED can save



(a) Cost (b) Delay

Figure 7.12: Event distance: 20, size: 15

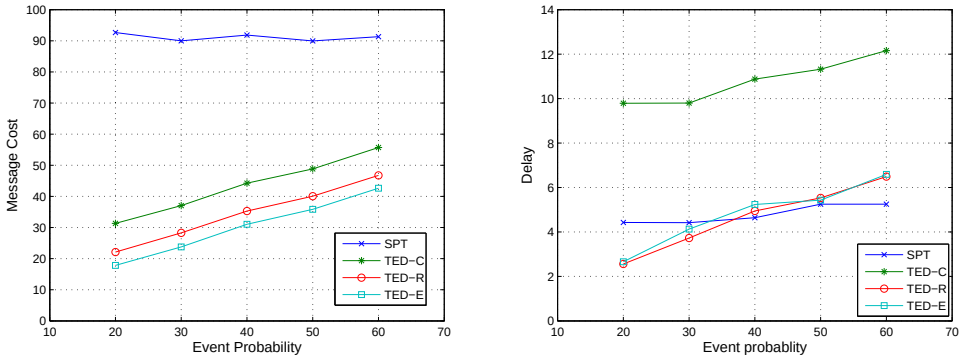
significant amount of message cost in comparison with SPT while introducing only marginal delay.

Our fifth set of results is shown in Figure 7.12. On the X-axis, the event probability spans from 20% to 60%. Note that for composite events, this is a conditional probability when all the sub-events happen. For instance, if a composite event e_3 comes from two sub-events e_1 and e_2. Then we have the following:

$$P(e_3) = P(e_1) \times P(e_2) \times P(e_3|e_1, e_2)$$

Our final set of results is shown in Figure 7.13. Similar to our previous sets of results, TED can save around 50% - 60% of energy cost compared with SPT while the delay has not much change.

In summary, we can find our simulation results agree with our analytical results. The message cost of TED is linearly proportional to the event probability while the message cost of SPT has little difference in regard to



(a) Cost (b) Delay

Figure 7.13: Event distance: 20, size: 20

probability changes. This characteristic will make TED especially effective for applications where the primitive events happen very frequently while the composite events happen rarely. We can also see that TED can help to reduce the delay when the event probability is low.

7.3 Experiments

As described in Section 7.1, the performance of TED heavily depends on the actual WSN applications because different applications may have different very different event definitions. In this section, we study some of thes WSN applications and find out how well TED performs.

We have implemented TED on MicaZ based on the existing middleware layer as in [60]. Similar to Section 7.1, we implemented a module that does opportunistic data aggregation based on the existing routing protocol provided by TinyOS for comparison.

We compare the performance using the following metrics:

(a) The sensor node (b) The light sensor

Figure 7.14: Car park sensor platform

- Message cost: this is obtained by setting up a counter inside the sensor node. The counter will be written into flash after the experiment so that we can retrieve it.

- Event detection delay: we measure the time between the subscription is disseminated and the event is notified.

7.3.1 Application Case One: Car Park

Our first application case is an intelligent car park [103]. We deployed some micaz sensor nodes for the application. For simplicity, we use light sensor to detect the presence of a vehicle. For better communication, the sensor nodes are attached close to the ceiling instead of on the ground. The light sensor on each node is connected through an extended cable as shown in Figure 7.14.

The floor plan and deployment of the sensor nodes is shown in Figure 7.15. The arrows represent the driving direction of the cars. Each letter 'p' represents a parking slot. In such a system, the management is interested in the number of park spaces and the location of them [103]. The primitive

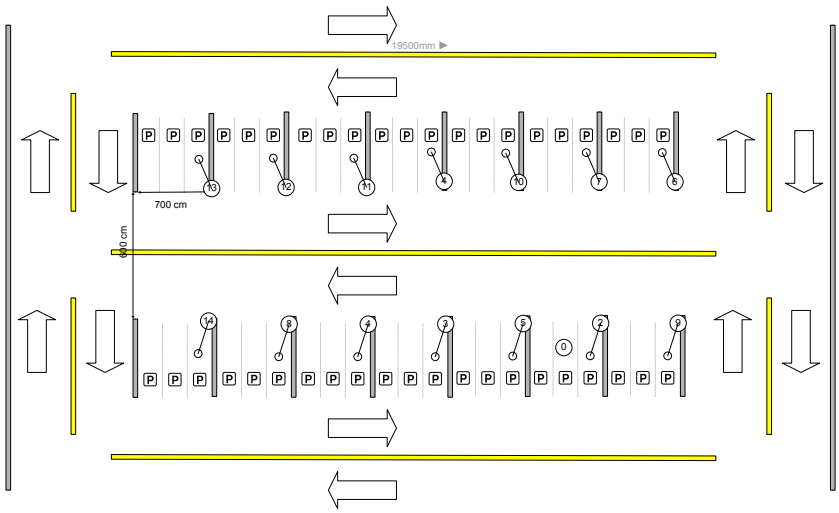
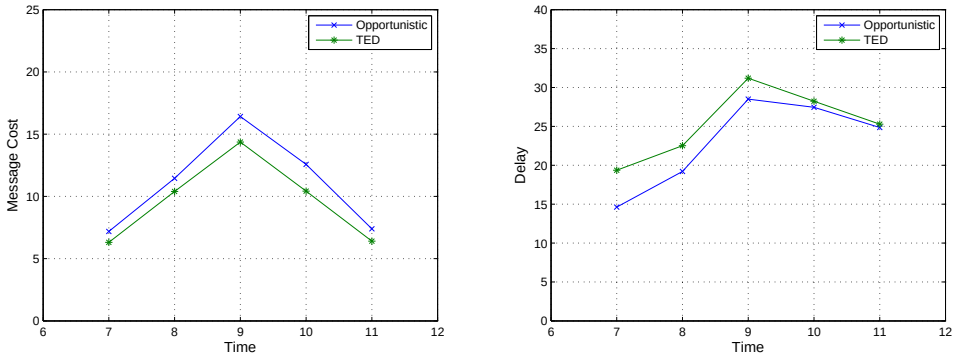


Figure 7.15: Car park sensor deployment

events for such a system will be the availability of individual car park spaces. Based on the primitive event, if we want to get notified when the parking spaces near the exit become available, then we just need to define composite events which locate the spaces with certain IDs. Th event definitions are shown in Listing 7.1. Here, the composite event takes two primitive events for parking space 1 and 2 which are close to the exit. The experimental results are shown in Figure 7.16. In this application, we primarily consider only the message costs because the delay isn't that important in such a system. The message cost is highest during rush hour when there are a lot of cars entering and leaving the car park. The message cost saved by TED is around 10-20% regardless of the total messages. Note that for both figures, since we did the experiments in the morning, there's a spike in the message cost during the rush hour.

Listing 7.1: Event definition for a car park

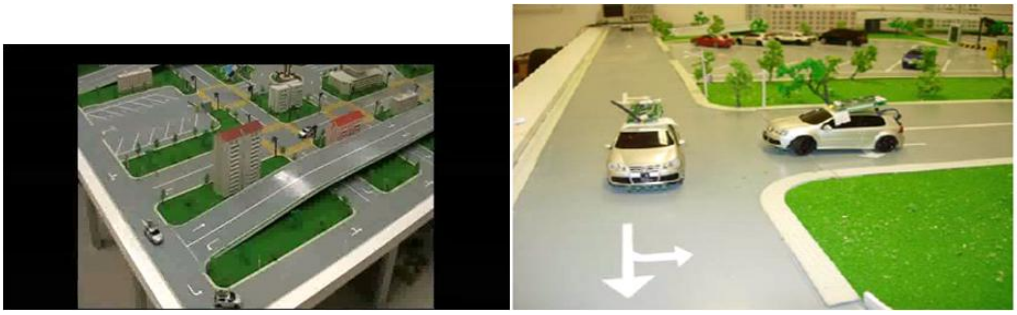
```
1 Event ParkSpaceEvent {
```



(a) With three fusion points (b) With four fusion points

Figure 7.16: Car park experiment results

```
2     int id=System.id;
3     int time=System.time;
4     int light=System.light;
5  } where {
6     light>THRESHOLD
7  }
8  Event CarParkEvent {
9     int id=System.id;
10 } on {
11    ParkSpaceEvent e1, e2;
12 } where {
13    e1.id==1 ||
14    e2.id==2 ||
15    e1.time-e2.time<10
16 }
```



(a) Our iTranSNet testbed (b) The model cars for iTranSNet testbed

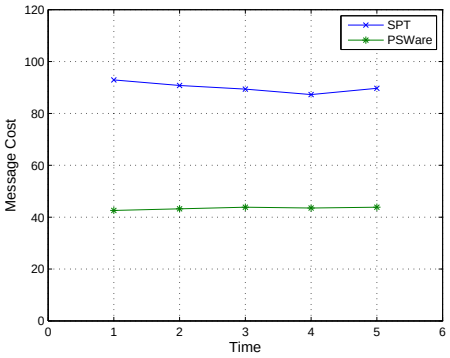
Figure 7.17: Lab testbed for transportation systems

7.3.2 Application Case Two: Transportation Systems

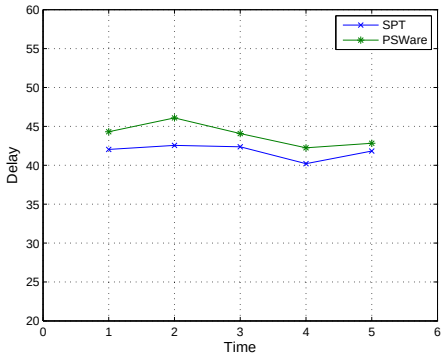
Apart from intelligent car park, another related application is WSN-based intelligent transportation system [62]. We also use the Micaz nodes to deploy such an application using our TED. Before we do the field test, we first perform some simple test on our lab testbed, iTranSNet. Figure 7.17 shows our testbed setup. In such testbed, we mainly use the following components to emulate a transportation system:

- Programmable model cars with UART interface to interact with MicaZ.
- Roads with light sensors to detect the presence of the model cars.
- Other programmable auxiliary facilities such as lamps and traffic lights.

Each road has magnetic strips underneath so that cars can be controlled and follow the road. In addition, IC cards have been installed near the intersections for each road. Each model car is equipped with a card reader so that when it comes close to the intersection, the program can make a decision on whether the car needs to turn or not.



(a) Message cost



(b) Delay

Figure 7.18: Experimental results on lab testbed: iTranSNet

In order to test different scenarios, we defined different event types for potentially different ITS applications. Listing 7.2 shows one of the most basic events for detecting a single vehicle.

Listing 7.2: Event definition for detecting a single vehicle

```
1 Event CarEvent {
2   int time=System.time;
3   int magnetic=System.magnetic;
4   int location=System.location;
5 } where {
6   magnetic>THRESHOLD
7 }
```

Listing 7.3 defines an event for detecting an over speeding vehicle.

Listing 7.3: Event definition for over speeding

```
1 Event SpeedEvent {
2   int speed=(e1.location-e2.location)/(e1.time-e2.time);
3 } on {
```

```
4    CarEvent e1, e2;
5  } where {
6    e1.time>e2.time &&
7    speed>THRESHOLD
8  }
```

Listing 7.4 defines an event for detecting a traffic jam.

Listing 7.4: Event definition for traffic jam

```
1  Event TrafficJam {
2    int count=count(e1);
3    int roadID=e1.roadID;
4  } on {
5    CarEvent e1;
6  } where {
7    roadID==CERTAIN_ROAD &&
8    count>THRESHOLD
9  }
```

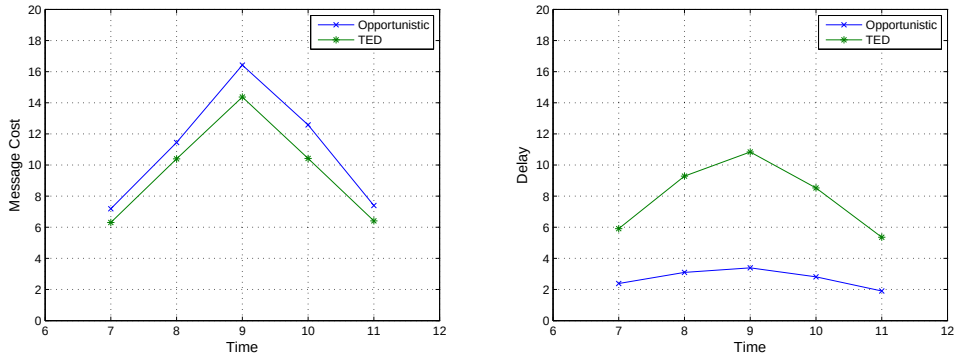
Different from the car park application, such applications are more delay sensitive. So we also measured the time delay for the event detection. We performed the testing on all the event types defined. The results are presented in Figure 7.18. The experimental results basically agree with the simulation results and show PSWare can save energy without too much delay.

After PSWare passes lab testing, we further deployed it in the real environment. The deployment in the real environment is shown in Figure 7.19.

The event definitions are the same as those we have used for our indoor test. The results are shown in Figure 7.20. Similar to the car park ap-

(a) Road side sensor nodes (b) Sensor nodes on the lamp

Figure 7.19: Sensor nodes for transportation systems



(a) Message cost (b) Delay

Figure 7.20: Experimental results on the real roads

plication, TED can save 10-20% energy while the delay is only a couple of milliseconds.

7.3.3 Application Case Three: Indoor Monitoring

Our third application is related to smart building. We consider the application scenario where the sensor nodes are deployed in a building so that the temperature can be monitored. Such an application can probably be useful

for certain types of context aware pervasive applications. For example, the air conditioner can be adjusted if several adjacent rooms' temperature rises too fast. The primitive and composite event definitions are shown in Listing 7.5.

Listing 7.5: Event definition for indoor monitoring

```
1  Event SingleTemp {
2    int id=System.id;
3    int temperature=System.temperature;
4  } where {
5    temperature>THRESHOLD
6  }
7  Event CompositeTemp {
8  } on {
9    SingleTemp e1, e2, e3;
10 } where {
11   e1.id==1 &&
12   e2.id==2 &&
13   e3.id==4
14 }
```

The primitive event simply tests if the temperature passes certain threshold and the composite event is the conjunction of several primitive events. We deployed the some Micaz nodes in different rooms in our building as shown in Figure 7.21. In the figure, each circle represents a sensor node.

The experimental results is shown in Figure 7.20. The results show that TED can save the message cost by 10-20%. Also note that in order to emulate certain events such as fire, we put some sensors on the heater during

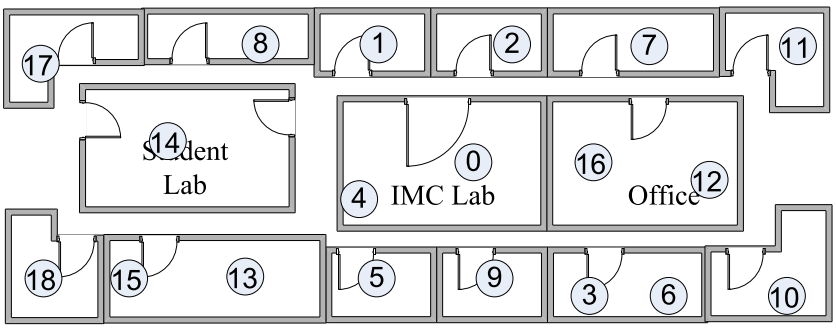


Figure 7.21: Deployment of the senosr nodes for indoor monitoring

the experiment so there's a spike in each figure.

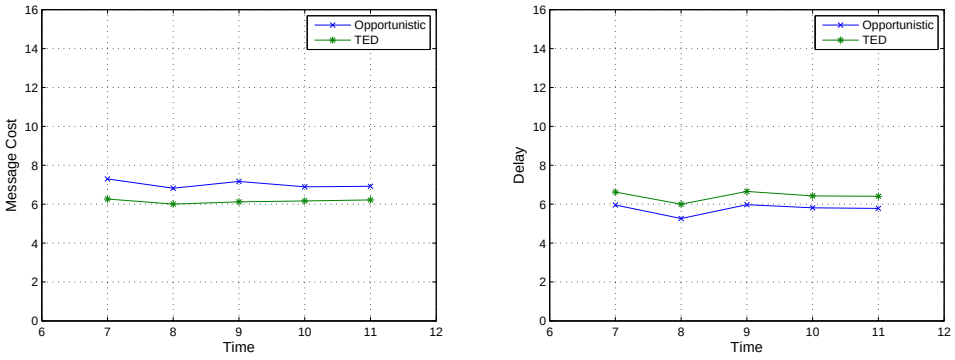
7.3.4 Application Case Four: SHM

Our final application is WSN-based Structural Health Monitoring (SHM) system which was described in Chapter 5. The objective of such system is to detect damages on structures such as buildings and bridges if they occur. Event detection is important in these applications because the SHM sensors will introduce high energy consumption during damage detection. It is therefore more desirable to wake them up only upon the occurrence of certain events [49].

Figure 7.23a shows our WSN-SHM testbed. In our experiment, we defined a scenario where the sensor nodes will start to collect the data when a certain vibration pattern is detected. The vibration is detected if the one sensor on the top and another one on the bottom of the model read the vibration data that satisfy certain criteria. The event definition is shown in Listing 7.6.

Listing 7.6: Event definition for SHM

```
1 Event Vibration {
```

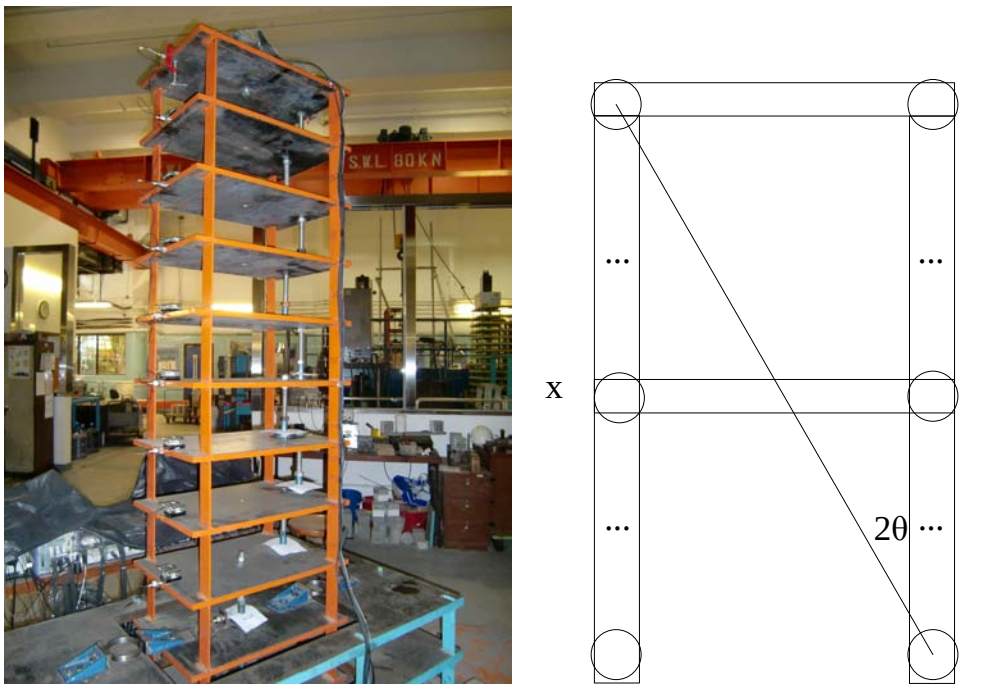


(a) With 2 fusion points (b) With 3 fusion points

Figure 7.22: Experiments for temperature monitoring

```
2     data=System.Vibration;
3   } where {
4     data>THRESHOLD1
5   }
6   Event CompVibration {
7   } on {
8     Vibration e1 and
9     Vibration e2
10  } where {
11    e1.location=='top' &&
12    e2.location=='bottom' &&
13    e1.data-e2.data>THRESHOLD2
14  }
```

With the help of our testbed, we can manually generate events by hitting the model in our testbed. Similar to our simulation, we implemented a naïve event detection method where the nodes simply use the existing routings provided by TinyOS [31] and detect events opportunistically. We implement

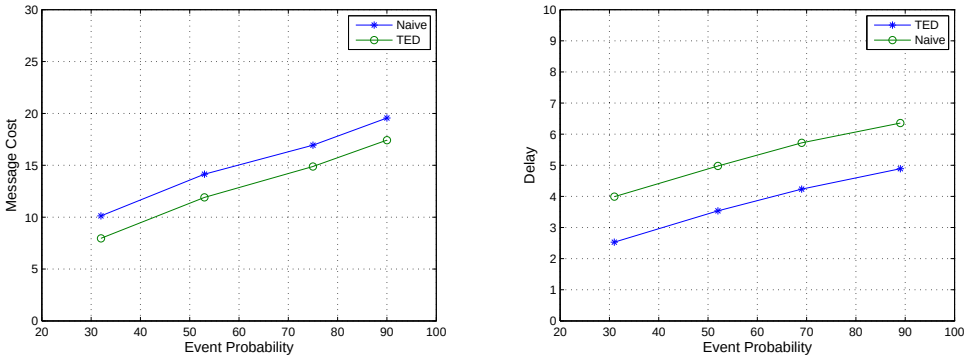


(a) The testbed (b) Parameters used in the experiments

Figure 7.23: PSWare experiment setup

two modules for both our TED and the opportunistic filtering where only the primitive events are filtered. In order to create multi-hop communication, we adjust the nodes communication range so that they can only communicate with nearby neighbors. For TED, we use tested both the centralized and the distributed versions and for the distributed version. For TED, we have the parameters set as shown in Figure 7.23b.

Figure 7.24 shows all our experimental results. In addition to energy efficiency, we also studied delay. We calculate the delay for TED as the duration between the time that naïve approach detects the event and the time that TED detects them. This is because naïve approach sends all the detected primitive events directly to the sink and the delay is only introduced by the multi-hop routing latency while both TED have additional delay in



(a) Message cost (b) Delay

Figure 7.24: Experimental results for SHM

detecting the events. The experimental results are expected because TED slightly introduces more delay when doing the fusion points selection. The results on message cost is similar to our simulations.

Chapter 8

Conclusion and Future Directions

8.1 Conclusion

In this work, we presented PSWare, a pub/sub middleware for WSN supporting composite events. We first described our research motivation for PSWare. Because of the common requirements in WSN-based applications for event detection, a middleware can significantly save the application developers' programming effort. To develop efficient PSWare, several research and engineering issues must be addressed. These issues include middleware abstraction, event definition, distributed composite event detection and system design.

We reviewed a lot of existing works that are related to the issues in our middleware. These existing works span several areas such as WSN-based middleware, macroprogramming in WSN, event based systems and

data aggregation for WSN. For each of the area, we categorized the existing works and compared their differences. We summarized the existing works and find out the room for improvement in PSWare.

PSWare was designed with all the above issues in mind. It uses a flexible architecture where different types of composite event detection algorithms can be easily integrated into it. Each layer represents the solution for a particular issue that we have found out.

Apart from the engineering issues, we did an extensive study on the research issues. The first is to study is the problem of general composite event detection for WSN. We proposed a novel distributed composite event detection algorithm, TED, for WSN. We proved that the composite event detection problem is NP-complete. Therefore, TED consists of a set of heuristic algorithms to forward the events and select fusion points. We have derived some important theorems regarding to the performance of TED.

The second research issue is clustering in our middleware. The issue was raised from one of the PSWare enabled application - SHM. We formulated the problem and proved it to be NP-complete. We proposed both centralized and distributed solutions to the clustering problem.

We have implemented our algorithms and PSWare in the real hardware sensor platform. We described our implementation approach. In addition, we built up some real WSN-based applications using PSWare. We evaluated the performance of PSWare through analysis, simulation and experiments. We compared the performance of PSWare with opportunistic data aggregation where events are aggregated without considering their event relations. By making use of the event fusion points, TED can detect composite events

in an energy efficient fashion. Many WSN-based applications can be easily developed with high efficiency.

8.2 Future Directions

Though PSWare has achieved good performance in event detection, there is still room for improvement. First, in PSWare, we have addressed the event definition and detection problems. However, subscription dissemination problem may also be an interesting issue. In this work, we haven't considered subscription dissemination problem because all the subscriptions are disseminated into the entire network. We think it is possible that further energy saving could be achieved if we jointly consider subscription dissemination and event detection. The system may select the event detectors according to the event subscriptions. Individual sensor nodes may order the subscriptions for better event fusion results. They may even partially evaluate the subscriptions to help selecting fusion points.

Another direction in the research issue is to consider the network dynamics. Currently, our fusion points selection algorithms and clustering algorithms are relatively static. Re-selection may be performed once the energy efficiency drops to certain threshold. This can be improved if the sensor nodes can perform re-selection or partial re-selection according to more criteria. Such problems may also be interesting optimization problems and are worth study.

Apart from research issues, there are other directions that may be further worked on. For example, we can deploy PSWare in a larger scaled application

scenario. Our current experiments in the real environments are still quite small. We can, for instance, deploy the senor nodes on a real bridge for SHM applications or deploy the sensor nodes in the real transportation networks with the traffic lights to make our results more convincing.

References

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 582–589, 2004.

[2] Hüseyin Akca and Hervé Brönnimann. A new deterministic data aggregation method for wireless sensor networks. *Signal Processing*, 87(12):2965–2977, December 2007.

[3] Lan F. Akyildiz, Welljan Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, August 2002.

[4] Guruduth Banavar, Tushar Ch, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–272, June 1999.

[5] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18, September 1999.

[6] S. Banerjee and S. Khuller. A clustering scheme for hierarchical control in multi-hop wireless networks. In *IEEE INFOCOM*, volume 2, pages 1028–1037. Citeseer, 2001.

[7] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowel Du, T.W. Danny Kim, , Blng Zhou, and Emin Gr̈(u)n Sirer. On the need for system-level support for ad hoc and sensor networks. *ACM SIGOPS Operating Systems Review*, 36(2):1–5, April 2002.

[8] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

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

[10] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, pages 3–14, 2001.

[11] Athanassios Boulis, Chih chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable

sensor networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Service*, pages 187–200, 2003.

[12] Francçis Bry and Michael Eckert. A high-level query language for events. In *Proceedings of the IEEE Services Computing Workshops*, pages 31–38, 2006.

[13] Jiannong Cao, Hejun Wu, Xuefeng Liu, and Yi Lai. isensnet: an infrastructure for research and development in wireless sensor networks. *Frontiers of Computer Science in China*, 4(3):339–353, 2010.

[14] Nicholas Carriero and David Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[15] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[16] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 606–617, 1994.

[17] Yuanzhu Peter Chen, Arthur L. Liestman, and Jiangchuan Liu. A hierarchical energy-efficient framework for data aggregation in wireless sensor networks. *IEEE Transactions on Vehicular Technology*, 55(3):789–796, May 2006.

[18] Simon Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 602–610, 2002.

[19] Crossbow. Crossbow website. `http://www.crossbow.com`.

[20] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering*, pages 261–270, 1998.

[21] Carlo Curino, Matteo Giani, Marco Giorgetta, and Alessandro Giusti. Tinylime: Bridging mobile and sensor networks through middleware. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 61–72, March 2005.

[22] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, 1997.

[23] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[24] Kai-Wei Fan, Sha Liu, and Prasun Sinha. Scalable data aggregation for dynamic events in sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 181–194, November 2006.

[25] Kai-Wei Fan, Sha Liu, and Prasun Sinha. Structure-free data aggregation in sensor networks. *IEEE Transactions on Mobile Computing*, 6(8):929–942, August 2007.

[26] C.R. Farrar and K. Worden. An introduction to structural health monitoring. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 365(1851):303, 2007.

[27] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions Autonomous Adaptive System Special Issue on Self-Adaptive and Self-Organizing Wireless Networking Systems*, 4(3):1–26, July 2009.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[29] Jie Gao, Leonidas Guibas, Nikola Milosavljevic, and John Hershberger. Sparse data aggregation in sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 430–439, May 2007.

[30] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. In *Proceedings of the first International Workshop on Rules in Database Systems*, 1993.

[31] David Gay, Matt Welsh, Philip Levis, Eric Brewer, Robert von Behren, and David Culler. The nesc language: A holistic approach to networked

embedded systems. In *Proceedings of Programming Language Design and Implementation*, pages 1–11, 2003.

[32] Andreas Geppert and Dimitrios Tombros. Event-based distributed workflow execution with eve. Technical report, University of Zurich, 1996.

[33] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, 2009.

[34] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 69–80, 2004.

[35] Robert E. Gruber, Balachander Krishnamurthy, and Euthimios Panagos. The architecture of the ready event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

[36] Chao Gui and Prasant Mohapatra. Power conservation and quality of surveillance in target tracking sensor networks. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, pages 129–143, November 2004.

[37] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In *Proceed-*

ings of the International Conference Distributed Computing in Sensor Systems, pages 126–140, June/July 2005.

[38] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 254–273, 2004.

[39] Salem Hadim and Nader Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 7(3):1–1, March 2006.

[40] Tian He, Brian M. Blum, John A Stankovic, and Tarek Abdelzaher. Aida: Adaptive application independent data aggregation in wireless sensor networks. *ACM Transactions on Embedded Computing System, Special issue on Dynamically Adaptable Embedded Systems*, 3(2):426–457, May 2003.

[41] Tian He, Lin Gu, Liqian Luo, Ting Yan, John A. Stankovic, and Sang H. Son. An overview of data aggregation architecture for real-time tracking with sensor networks. In *20th International Parallel and Distributed Processing Symposium*, pages 1–8, April 2006.

[42] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 146–159, 2001.

[43] Wendi B. Heinzelman, Anantha P. Chandrakasan, and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 40(8):660–670, October 2002.

[44] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perilloe. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, June 2004.

[45] Karen Henricksen and Ricky Robinson. A survey of middleware for sensor networks: State-of-the-art and future directions. In *Proceedings of the International Workshop on Middleware for Sensor Networks*, pages 60–65, 2006.

[46] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 81–94, 2004.

[47] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: A distributed mobile sensor computing system. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems*, pages 125–138, 2006.

[48] G.H. James III, T.G. Carne, and J.P. Lauffer. The natural excitation technique (NExT) for modal parameter extraction from operating wind turbines. *NASA STI/Recon Technical Report N*, 93:28603, 1993.

[49] ShinAe Jang, Hongki Jo, Soojin Cho, Kirill Mechitov, Jennifer Rice, Sung-Han Sim, Hyung-Jo Jung, Chung-Bang Yun, B. F. Spencer, and Gul Agha. Structural health monitoring of a cable-stayed bridge using smart sensor technology: Deployment and evaluation. *Smart Structures and Systems*, 6(5):439–460, July 2010.

[50] Jaein Jeong, Sukun Kim, and Alan Broad. Xnp documentation. `http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf`, August 2003.

[51] Hongbo Jiang and Shudong Jin. Scalable and robust aggregation techniques for extracting statistical information in sensor networks. In *Proceedings of the 26th IEEE International Conference on Distributed Computing System*, page 69, 2006.

[52] Porlin Kang, Cristian Borcea, Gang Xu, Akhilesh Saxena, Ulrich Kremer, and Liviu Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal*, 47(4):475–494, 2004.

[53] Brad Karp and H. T. Kung. Gpsr: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pages 243–254, 2000.

[54] Lawrence A. Klein. Traffic parameter measurement technology evaluation. In *Proceedings of the IEEE-IEE Vehicle Navigation and Information Systems Conference*, pages 529–533, August 1993.

[55] Bhaskar Krishnamachari, Deborah Estrin, and Stephen B. Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 575–578, 2002.

[56] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[57] Steven Lai, Jiannong Cao, and Xiaopeng Fan. Ted: Efficient type-based composite event detection for wireless sensor network. In *Proceedings of 7th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'11)*, June 2011.

[58] Steven Lai, Jiannong Cao, and Xiaopeng Fan. Ted: Efficient type-based composite event detection for wireless sensor network. In *Proceedings of 7th IEEE International Conference on Distributed Computing in Sensor Systems*, June 2011.

[59] Steven Lai, Jiannong Cao, and Yuan Zheng. Psware: A publish / subscribe middleware supporting composite event in wireless sensor network. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications (PerSeNS'09)*, pages 1–6, March 2009.

[60] Steven Lai, Jiannong Cao, and Yuan Zheng. Psware: A publish / subscribe middleware supporting composite event in wireless sensor

network. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, pages 1–6, March 2009.

[61] Yi Lai, Yuan Zheng, and Jiannong Cao. Protocols for traffic safety using wireless sensor network. In *Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'07)*, pages 37–48, June 2007.

[62] Yi Lai, Yuan Zheng, and Jiannong Cao. Protocols for traffic safety using wireless sensor network. In *Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*, pages 37–48, June 2007.

[63] Marc Lee and Vincent W.S. Wong. Lpt for data aggregation in wireless sensor networks. In *IEEE Global Telecommunications Conference*, pages 2969–2974, December 2005.

[64] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review*, 36(5):85–95, December 2002.

[65] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems*, pages 126–137, November 2003.

[66] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei. Event detection services using data service middleware in distributed

sensor networks. *Telecommunication Systems*, 26(2-4):351–368, June 2004.

[67] Stephanie Lindsey, Cauligi Raghavendra, and Krishna M. Sivalingam. Data gathering algorithms in sensor networks using energy metrics. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):924–935, September 2002.

[68] G. Liu, A.K. Mok, and E.J. Yang. Composite events for network event correlation. In *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, pages 247–260, May 1999.

[69] Jie Liu, Maurice Chu, Juan Liu, James Reich, and Feng Zhao. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing*, 2(4):50–62, October 2003.

[70] Jie Liu and Feng Zhao. Towards semantic services for sensor-rich information systems. In *Proceedings of the 2nd International Conference on Broadband Networks*, pages 967–974, October 2005.

[71] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118, 2003.

[72] Xuefeng Liu, Jiannong Cao, Md. Zakirul Alam Bhuiyan, Steven Lai, Hejun Wu, and Guojun Wang. Fault tolerant wsn-based structural health monitoring. In *Proceedings of 41st Annual IEEE/IFIP Inter-*

national Conference on Dependable Systems and Networks (DSN'11), June 2011.

[73] Xuefeng Liu, Jiannong Cao, Steven Lai, Chao Yang, Hejun Wu, and Youlin Xu. Energy efficient clustering for wsn-based structural health monitoring. In *Proceedings of 30th IEEE International Conference on Computer Communications (INFOCOM'11)*, April 2011.

[74] Benny Lo, Surapa Thiemjarus, Rachel King, and Guang Zhong Yang. Body sensor network - a wireless sensor platform for pervasive healthcare monitoring. In *The 3rd International Conference on Pervasive Computing*, pages 77–80, May 2005.

[75] Hong Luo, Yonghe Liu, and Sajal K. Das. Routing correlated data with fusion cost in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 5(11):1620–1632, November 2006.

[76] Jerome P. Lynch and Kenneth J. Loh. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest*, pages 91–128, 2005.

[77] J.P. Lynch, A. Sundararajan, K.H. Law, A.S. Kiremidjian, T. Kenny, and E. Carryer. Embedment of structural monitoring algorithms in a wireless sensing unit. *Structural Engineering and Mechanics*, 15(3):285–297, 2003.

[78] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *ACM SIGOPS Operating Systems Review*, pages 131–146, 2002.

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

[80] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, and David Culler. Wireless sensor networks for habitat monitoring. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, pages 88–97, June 2002.

[81] Masoud Mansouri-samani, Morris Sloman, and Morris Sloman. Gem - a generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[82] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. Tinycubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks*, pages 278–289, January 2005.

[83] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 167–180, 2006.

[84] Mohammad M. Molla and Sheikh Iqbal Ahamed. A survey of middleware for sensor network and challenges. In *Proceedings of the 2006*

International Conference Workshops on Parallel Processing, pages 223–228, 2006.

[85] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. In *IEE Proceedings - Software*, pages 1–10, 2001.

[86] Gero Mühl, Ludger Fiege, Alejandro Buchmann, Ludger Fiege Alej, and Ro Buchmann. Filter similarities in content-based publish/subscribe systems. In *Proceedings of the International Conference on Architecture of Computing Systems*, pages 224–238, 2002.

[87] T. Nagayama and B.F. Spencer Jr. Structural health monitoring using smart sensors. *Newmark Structural Engineering Laboratory Report Series 001*, 2008.

[88] Ryan Newton, Arvind, and Matt Welsh. Building up to macroprogramming: an intermediate language for sensor networks. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, pages 78–87, August 2005.

[89] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceeedings of the 1st International Workshop on Data Management for Sensor Networks in Conjunction with VLDB*, pages 78–87, August 2004.

[90] Math Pages. Mean distance from vertex to interior of plane figures. `http://www.mathpages.com/home/kmath283/kmath283.htm`.

[91] Animesh Pathak, Luca Mottola, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Proceedings of the Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 255–260, March 2007.

[92] Sundeep Pattem, Bhaskar Krishnamachari, R. Godindan, and Ramesh Govindan. The impact of spatial correlation on routing with compression in wireless sensor networks. In *Third International Symposium on Information Processing in Sensor Networks*, pages 28–35, April 2004.

[93] C. Perkins, E. Belding-Royer, and S.Das. Ad hoc on-demand distance vector (aodv) routing, 2003.

[94] Neil Patel Philip Levis, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, 2004.

[95] Vaskar Raychoudhury, Jiannong Cao, Weigang Wu, and Steven Lai. K-directory community: Reliable service discovery in manet. *Journal of Pervasive and Mobile Computing (JPMC)*, 7(1), February 2011.

[96] Kay Römer. Programming paradigms and middleware for sensor networks. In *GI/ITG Workshop on Sensor Networks*, pages 49–54, February 2004.

[97] Bill Segall and David Arnold. Elvin has left the building: A publish/-subscribe notification service with quenching. In *Proceedings of the 1997 Australian UNIX and Open Systems Users Group Conference*, 1997.

[98] Mohamed A. Sharaf, Jonathan Beaver, Alexandros Labrinidis, Ros Labrinidis, and Panos K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *The VLDB Journal - The International Journal on Very Large Data Bases*, 13(4):384–403, December 2004.

[99] Chien-Chung Shen, Chavalit Srisathapornphat, and Chaiporn Jaikaeo. Sensor information networking architecture and applications. *IEEE Personal Communications*, 8(4):52–59, August 2001.

[100] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: New aggregation techniques. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 239–249, 2004.

[101] Kannan Srinivasan and Philip Levis. Rssi is under appreciated. In *Proceedings of the Third Workshop on Embedded Networked Sensors*, 2006.

[102] Ramanan Subramanian, Hossein Pishro-Nik, and Faramarz Fekri. Clustering-based correlation aware data aggregation for distributed sensor networks. In *IEEE Global Telecommunications Conference*, pages 3253–3257, December 2005.

[103] Vanessa W.S. Tang, Yuan Zheng, and Jiannong Cao. An intelligent car park management system based on wireless sensor networks. In *Proceedings of the 1st International Symposium on Pervasive Computing and Applications*, pages 65–70, August 2006.

[104] P.J. Wan, K.M. Alzoubi, and O. Frieder. Distributed construction of connected dominating set in wireless ad hoc networks. *Mobile Networks and Applications*, 9(2):141–149, 2004.

[105] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, pages 3–3, 2004.

[106] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, pages 99–110, 2004.

[107] Yuan Xue, Yi Cui, and Klara Nahrstedt. Maximizing lifetime for data aggregation in wireless sensor networks. *Mobile Networks and Applications*, 10(6):853–864, December 2005.

[108] Eiko Yoneki and Jean Bacon. Unified semantics for event correlation over time and space in hybrid network environments. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 366–384, November 2005.

[109] Ossama Younis and Sonia Fahmy. An experimental study of routing and data aggregation in sensor networks. In *Proceedings of the IEEE*

International Conference on Mobile Ad-hoc and Sensor Systems, pages 57–65, November 2005.

[110] Ossama Younis, Marwan Krunz, and Srinivasan Ramasubramanian. Node clustering in wireless sensor networks: Recent developments and deployment challenges. *IEEE Network*, 20(3):20–25, May/June 2006.

[111] Yang Yu, Bhaskar Krishnamachari, and Viktor K. Prasanna. Issues in designing middleware for wireless sensor networks. *IEEE Network*, 18(1):15–21, January/February 2004.

[112] Wensheng Zhang and Guohong Cao. Dctc: Dynamic convoy tree-based collaboration for target tracking in sensor networks. *IEEE Transactions on Wireless Communication*, 3(5):1689–1701, September 2004.

[113] Jin Zhu, Symeon Papavassiliou, and Jie Yang. Adaptive localized qos-constrained data aggregation and processing in distributed sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):923–933, September 2006.

Appendix A

Complete List of the EDF Instructions

Here we show the complete list of instructions currently used by our Event Detection Framework (EDF). We also briefly describe the function and the usage of each instruction.

A.1 Basic Instructions

The basic instruction deals with the most fundamental operations. Since the instruction uses a stack-based architecture in order to reduce the code size, most of the instructions that fall into this category deal with the operations related stacks.

- *OPpush*: this instruction is used to push an operand to the top of the stack.
- *OPpop*: this instruction is used to pop and discard an operand from

the top of the stack.

- *OPcopy*: duplicates the top operand of the stack
- *OPhalt*: once this instruction is reached, the VM will stop execution.

A.2 Operators

Basic mathematical operators:

- *OPadd*: pops two operands off the top of the stack, add them together and push the result back to the top of the stack.
- *OPmult*: pops two operands off the top of the stack, multiply them together and push the result back to the top of the stack.
- *OPsub*: pops two operands off the top of the stack, subtract the first popped number by the second one and push the result back to the top of the stack.
- *OPdiv*: pops two operands off the top of the stack, divide the first popped number by the second one and push the result back to the top of the stack.
- *OPmod*: pops two operands off the top of the stack, divide the first popped number by the second one and push the remainder back to the top of the stack.
- *OPinv*: pops one operand (n) off the top of the stack, calculate its inverse (-n) and push the result back.

Logical operators:

- *OPand*: pops two operands off the top of the stack, calculate the logical and of them and push the result back to the top of the stack.
- *OPor*: pops two operands off the top of the stack, calculate the logical or of them and push the result back to the top of the stack.
- *OPxor*: pops two operands off the top of the stack, calculate the exclusive or of them and push the result back to the top of the stack.
- *OPnot*: pops one operand off the top of the stack, calculate the logical not of it and push the result back to the top of the stack.

Relational operators:

- *OPeq*: pops two operands off the top of the stack, if the two operands are equal, push 1 to stack. Otherwise, push 0 to stack.
- *OPneq*: pops two operands off the top of the stack, if the two operands are equal, push 0 to stack. Otherwise, push 1 to stack.
- *OPgt*: pops two operands off the top of the stack, if the first operand is greater than the second one, push 0 to stack. Otherwise, push 1 to stack.
- *OPgte*: pops two operands off the top of the stack, if the first operand is greater than or equal to the second one, push 0 to stack. Otherwise, push 1 to stack.
- *OPlt*: pops two operands off the top of the stack, if the first operand is less than the second one, push 0 to stack. Otherwise, push 1 to stack.

- *OPlte*: pops two operands off the top of the stack, if the first operand is less than or equal to the second one, push 0 to stack. Otherwise, push 1 to stack.

Bitwise operators:

- *OPshiftl*: pops two operands off the top of the stack, left shift the first popped number by the second one and push the result back to the top of the stack.
- *OPshiftr*: pops two operands off the top of the stack, right shift the first popped number by the second one and push the result back to the top of the stack.
- *OPland*: pops two operands off the top of the stack, calculate the bitwise and of them and push the result back to the top of the stack.
- *OPlor*: pops two operands off the top of the stack, calculate the bitwise or of them and push the result back to the top of the stack.
- *OPxor*: pops two operands off the top of the stack, calculate the bitwise exclusive or of them and push the result back to the top of the stack.
- *OPlnot*: pops one operand off the top of the stack, calculate its complement and push the result back to the top of the stack.

A.3 Event-related Instructions

- *OPinstall*: this instruction is invoked after the subscription is successfully installed on a sensor node.

- *OPref*: whenever an event e is being evaluated, this instruction is invoked to obtain an instance of such event.
- *OPoffset*: this instruction is involved right after the 'ref' instruction, in order to access individual attributes of the event instance.
- *OPset*: if the attributes of an event need to be changed, this instruction will be used.
- *OPget*: the 'get' instruction does the opposite of 'set' instruction. It will simply retrieve content of a specific attribute in an event.
- *OPcreate*: is used to create a new instance of an event.
- *OPeval*: is used to determine if an event happens or not.
- *OPgc*: used by the event matcher for garbage collection.