A SENSOR-BASED APPROACH TO MONITORING WEB SERVICE

by

Jun Li

A thesis submitted to the School of Computing

In conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

(November, 2008)

Abstract

As the use of Web expands, Web Service is gradually becoming the basic system infrastructure. However, as it matures and a large number of Web Service becomes available, the focus will shift from service development to service management. One key component in management systems is monitoring. The growing complexity of Web Service platforms and their dynamically varying workloads make manually monitoring them a demanding task. Therefore monitoring tools are required to support the management efforts.

Our approach, Web Service Monitoring System (WSMS), utilizes Autonomic Computing technology to monitor Web Service for an automated manager. WSMS correlates lower level events into a meaningful diagnosed symptom which provides higher level information for problem determination. It also gains the ability to take autonomic actions and solve the original problem using corrective actions. In this thesis, a complete design of WSMS is presented along with a practical implementation showing viability and proof of concept of WSMS.

Acknowledgements

I would like to thank my supervisor Dr. Patrick Martin and Wendy Powley for their constant support, advice and guidance throughout my research. I would also like to thank them for giving me the freedom to strive for innovation.

Chris Craddock, Serge Mankovski and Kirk Wilson from CA INC are also appreciated for their suggestions.

Credits also go to Queen's Database Systems Laboratory and the students there. I would like to thank my colleagues for their invaluable input and discussions. I especially would like to thank to Imad Abdallah, Cyrus Boadway and Vennie So for their help in testing the prototype.

Finally I would like to thank to my wife, Jing Shen, for supporting my study here at Queen's to pursue degree of Master of Science. And I would like to thank my parents to allow me to have my dream come true.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AWSE	Autonomic Web Services Environment
BPEL4WS	Business Process Execution Language for Web Service
BtM	Build to Manage
CEP	Complex Event Processing
DBMS	Database Management System
EPL	Event Pattern Language
EPR	End Point Reference
FTP	File Transfer Protocol
GF	Global Flow
GUI	Graphic User Interface
HTTP	Hypertext Transfer Protocol
JDBC	Java Database Connectivity
JEE	Java Enterprise Edition
JMS	Java Messaging Service
JVM	Java Virtual Machine
MAPE	Monitor, Analyze, Plan and Execute
MOWS	Management of Web Service
MUWS	Management Using Web Service
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SOA	Service Oriented Architecture

SOAP	Simple Object Access Protocol
TPTP	Test and Performance Tools Platform
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WEF	Web Service Distributed Management Event Format
WS	Web Service
WSDL	Web Service Description Language
WSDM	Web Service Distributed Management
WSMS	Web Service Monitoring System
WSN	Web Service Notification
WSA	Web Service Addressing
WSLA	Web Service Level Agreement
WSRF	Web Service Resource Framework

Chapter 1

Introduction

For decades, information technology (IT) has produced complex systems that challenge IT professionals. The complexity of these systems, and in particular the complexity of their management, is becoming a significant limiting factor in their further development. Large companies and institutions develop large-scale computer networks for communication and computation. The applications running on these distributed networks deal with many diverse tasks, ranging from controlling internal workflows to presenting web front ends and providing customer service [31].

Autonomic Computing [9] provides the technology to build self-managing IT infrastructures, including both hardware and software that can configure, heal, optimize, and protect themselves. An Autonomic Computing solution typically implements a feedback loop of system monitoring, system analysis, planning for action, and action execution. When the monitoring subsystem detects one or more events that indicate a decline in performance, action may need to be taken such as reallocation of resources, tuning of one or more components, or perhaps enforcement of workload control measures.

Web Service is a software model designed to support interoperable machine to machine interaction over a network. It provides a simplified method to connect systems regardless of the technologies or devices they use, or their locations. They are standard protocols supported by vendors which can leverage the internet for low cost communications, as well as other transport mechanisms such as HTTP and FTP protocols. The loosely coupled messaging approach reduces the cost of maintenance, the impact of changes and facilitates reuse of existing assets [19].

A Web Service environment consists of multiple distributed components running on heterogeneous platforms with multiple applications interacting in unpredictable ways. Such a complex environment is impossible to manage manually and the use of Autonomic Computing for the management of a Web Service environment has been proposed [30]. In this thesis, we propose a monitoring subsystem for an autonomic Web Service environment, namely Web Service Monitoring System (WSMS). WSMS monitors the system by collecting and observing events that indicate situations to be reacted to or problems to be diagnosed. When WSMS detects certain patterns of events, it informs an external system to take corrective action.

For example, in a Web Service environment, we could have a Web server responding to users' requests and a database system to store users' profiles. If the Web server experiences an increase in workload intensity, the response time may increase. In order to recover the performance, the Web server may need to be tuned to allow for more connections, or perhaps the database system may require additional memory. WSMS resides between the monitored Web server and the manager to provide the information needed to make decisions.

1.1 Motivation

With the magnificent explosion of Internet and Web Service, driven by the increasing requirement of different services, it is a challenge to manage software and services effectively. The complexity of the IT components as well as the relationships among them help fuel this problem. Besides, system failures, hardware and software issues and human errors impede system administration. Human intervention is required. However, it is driving up overall costs.

Automated management makes IT infrastructure and applications run more efficiently with less human intervention. For instance, an automated manager could automatically deploy a new resource, such as a new server, and tune the server's configuration for its intended usage. This is a

significant shift from traditional mechanisms that require a significant amount of manual intervention to ensure the resource operates effectively and efficiently, which can enhance the organization's ability to react to changes.

A monitoring system plays a key role in automated management. It is vital for management entity to obtain timely and accurate knowledge of the global environment. Fast detection of failures and malicious attacks is becoming one of the most important missions for monitoring. Those issues must be detected before any further actions are taken by either human administrators or decision-making systems. Further, information of the environment from different perspective is also required to feed different applications. To solve this problem, a lot of solutions for managing information systems have emerged. In 1990, Simple Network Management Protocol (SNMP) [4] was designed for managing and monitoring networks. Swatch [23] is an approach an approach to monitoring events on a large number of servers and workstations. IBM's Autonomic Computing [9] technology includes a monitoring component to assist in problem determination.

It is convenient if we are able to monitor and analyze events directly related to problems that are likely to arise. An event is an object that is a record of an activity in a system [15]. It can be expressed by metrics using plain text or another format. In a distributed environment we can say that events generated by multiple systems form event clouds [15]. An event cloud is a partially ordered set of events, either bounded or unbounded, where the partial orderings are imposed by the causal, timing and other relationships between the events [15]. A group or a sequence of correlated events may represent potential problems. For example, we have a Web server hosting a website. It keeps sending events with the current performance metrics to a manager. It works fine until the workload becomes much higher caused by intensive accesses. Correspondingly, the

metrics data in the events show fluctuations. Therefore, by finding the fluctuations included in a sequence of events we can detect the change in the Web server.

1.2 Objective

The main objective of this thesis is to design and implement a flexible monitoring system in an autonomic Web Service environment. The system is capable of monitoring individual, possibly physically distributed components, correlating events arising from these components and looking for certain patterns of events that may be indicative of potential problems. It is composed of lightweight entities, which we call Sensors. A Sensor is a Web Service component to consume events, evaluate patterns against them and produce complex events. A group of Sensors forms a hierarchy to interpret low level events and translate them into more meaningful higher level events that can be understood by decision-making systems or a management entity.

Furthermore, the monitoring system works in a Web Service Distributed Management (WSDM) [12] environment. The way events are sent and received is defined by Web Service Notification framework [6]. In order to consume and produce events, the monitoring system must comply with it. Additionally, WSDM Event Format is the schema for messages flowing in WSDM environment. So, our monitoring system is able to parse this schema.

WSMS is a lightweight, flexible, standards-based approach to monitoring in an autonomic system. Although our focus is on monitoring in a Web Service environment, our approach can be applied to any distributed system that is capable of producing events. Our work combines Complex Event Processing (CEP) [15] with Web Service technology and, to the best of our knowledge, there is no similar research work in the available literature.

1.3 Contribution

This thesis designs and implements a monitor for an Autonomic Web Service Environment. It collects information from monitored systems, processes them and outputs symptoms accordingly to management entity for further decisions or actions. It is a system capable of detecting suspicious behaviors automatically. Moreover, it enhances and extends Web Service systems with an advanced technology, CEP. More specifically, it analyzes events transferring among Web Service systems by correlating them in a time manner. Last but not the least, it is a system based on policy. By writing a custom implementation of a given policy, our system can be used in situations requiring behaviors unforeseen by us currently. This feature contributes to high flexibilities and extensibility of our system.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 provides the background and related work. Three key technologies are identified: WSDM, CEP, and Autonomic Computing. We provide an overview of Autonomic Web Service Environment (AWSE) [30], a framework for autonomic management in a Web Service environment which we use as our test bed. Chapter 3 presents the architecture and detailed design of the monitoring system. Chapter 4 describes a prototype implementation of our monitoring system integrated with AWSE and we present an evaluation scenario that demonstrates the scope of the monitoring system's capabilities. Chapter 5 provides conclusions, the limitations of our monitoring system, and suggestions for future work.

Chapter 2
Background and Related Work

Our approach utilizes three key technologies: Autonomic Computing, Web Service and CEP. In this chapter, we introduce these technologies followed by related work.

2.1 Autonomic Computing

Autonomic Computing aims to create computer systems capable of self-management in order to overcome the rapidly growing system complexity [9]. By taking care of many of the ever growing management requirements of IT systems, Autonomic Computing allows people to focus on business issues. The IT industry has been growing exponentially for decades. The increasing expense of IT maintenance is a big issue for the business world. In order to tackle this problem, IBM proposed Autonomic Computing [9] which is based on the notion that systems can be built to mimic the human autonomic nervous system which regulates most functions without conscious effort. Figure 2-1 shows the basic Autonomic Computing feedback loop, often called the MAPE loop that includes four components: Monitor, Analyzer, Planner and Executer.

The monitor component collects, aggregates, filters and reports details (such as metrics and performance data) collected from a managed system so that it is able to provide symptoms for the analyzer. The analyzer component correlates and models complex situations. These methods allow the manager to learn about the IT environment and help determine problems or predict future situations. The planner component constructs the actions to be taken to achieve goals and objectives. The executor component executes a plan of action to reconfigure or adjust the managed system. The knowledge base is a registry, dictionary, database or other repository that provides access to knowledge.

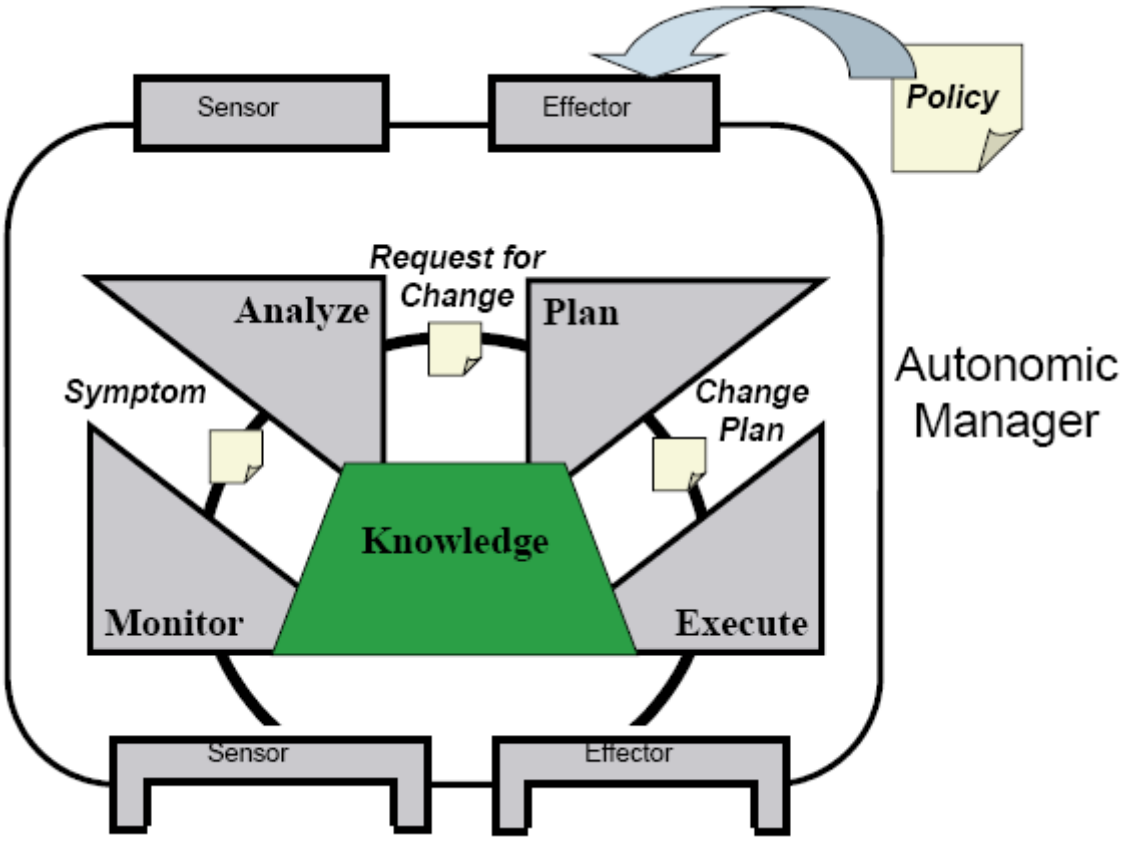

Figure 2-1 MAPE Loop in Autonomic Computing (Reproduced with permission [8])

2.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is a technology for aggregating various capabilities that may be controlled by different managers [32]. In general, service entities provide capabilities to compose services such as executing a workflow, submitting a query, returning results to a request, and so forth.

One of the main goals of SOA-enabled systems is to bring together fairly large chunks of capabilities from existing software services to form ad hoc applications. It increases reusability and flexibility as well as reduces costs in business to business cooperation.

SOA services are loosely coupled so that the implementation is simplified because it is hidden from the caller. Therefore, SOA can be developed using traditional languages such as JAVA, C#, COBOL or PHP. XML has been used extensively to describe services functionality and wrap interaction messages between services.

2.3 Web Service

Web Service [31] is a software model designed to support interoperable machine to machine interaction over a network. It can be used to implement a service-oriented architecture in the Web environment. Web Service emerged in 2000, and standards were proposed in the same year. Generally, Web Service is a set of Web APIs which can be accessed over Web, and be executed on a remote system hosting the requested services. For example, Amazon provides a Web Service called Amazon Simple Storage Service. It allows users to store and retrieve data from anywhere on the Web. Figure 2-2 shows a typical way to engage a Web Service.

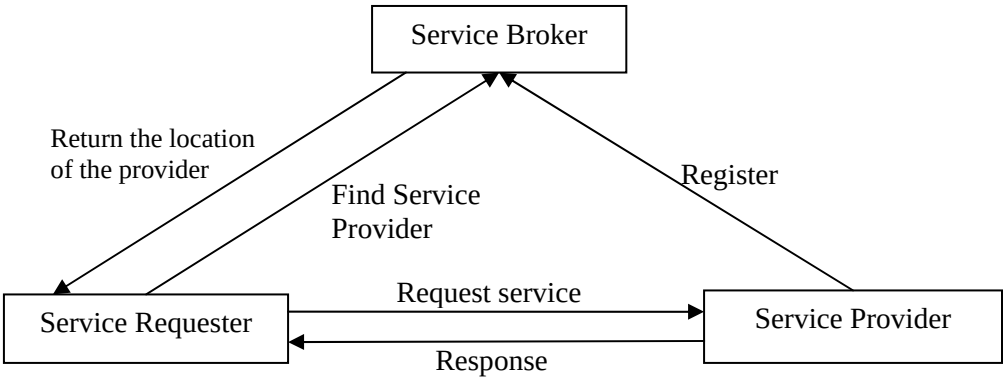

Figure 2-2 Engaging a Web Service

In this model, the Service Provider registers the service at the Service Broker so that the Service Requester is able to find the location of Service Provider at the Service Broker. Then the Service Requester sends the request to the Service Provider for services and the Service Provider

responds to the requester with the requested services. Web Service tries to maximize the interoperability and minimize the degree of coupling between a requester and a provider. Therefore, the Service Provider becomes reusable and easy to extend.

There are abundant standards pertaining to Web Service. The basic standards include Universal Description Discovery and Integration (UDDI) [3] for publishing and automatic service discovery, Web Service Description Language (WSDL) [5] for describing and invoking services in a unified way, and Simple Object Access Protocol (SOAP) [7] for exchanging information in XML format. In addition to the above three fundamental standards, we may categorize Web Service protocols into the following areas: Security, Reliable Messaging, Transactions, Messaging, Metadata, XML and Management. For example, WS-Notification (WSN) [6] and WS-Addressing (WSA) [2] are used to represent and locate services, respectively. WSDM [18] is a standard for management.

The WSA standard [2] defines mechanisms to address Web Service and messages flowing within it. It detaches the dependency on a transport protocol from addressing services. In other words, it can rely on any messaging platform or transport layer. With this strategy, messages representing events that happen in a Web Service environment, based on the address reference within the message body, flow to the corresponding endpoint, which represents a Web Service entity.

The WSN framework [6] is based on event notification. It defines a mechanism for a service to distribute information to other services, without prior knowledge of the receivers. WSN includes two sub-specifications: WS-BaseNotification and WS-Topic. WS-BaseNotification defines the interface WS-Notification consumers and producers should expose. WS-Topics addresses how to define and use topics to which Web Service consumers can subscribe, as well as organize them into complex structures such as hierarchies (topic trees) and synonyms (topic aliasing).

2.4 Web Service Distributed Management

WSDM [18] is a Web Service standard for managing other services. It allows a manager service to communicate with any other service that is WSDM-compliant. For example, a third-party management console can be used to observe the status or performance of a group of printers, and potentially send an alarm when a printer is out of toner.

WSDM consists of two specifications, namely Management Using Web Service (MUWS) [12] and Management of Web Service (MOWS) [13]. MUWS defines the interfaces to represent and access the manageable functions of Web Service resources. The manageability capabilities include resource identity, reporter identity, situation, and relationships, which can be composed together to represent the capability of the management metrics. MUWS also provides a standard management event format, namely WSDM Event Format (WEF) to improve interoperability and correlation between events. MOWS defines how to manage Web Service resources as well as how to describe and access the manageable functions using MUWS. MOWS also defines how manageable Web Service applications interoperate across enterprise and organizational boundaries.

WSDM supports notifications using WSN event format and WEF messages. WSN provides the publish-subscription services for Web Service architectures, as we describe in Section 2.3.

2.5 WSDM Event Format

The WSDM Event Format is an extensible XML format that defines a set of fundamental elements that allow different types of management information to be represented in a regulated expression. WEF includes elements indicating situations and metadata about event sources. As we can see in the schema for WEF in Appendix A, SourceComponent identifies where events happen; ReporterComponent identifies the component reporting the events (which may not be the

same as the source component) and the Situation element represents event details, such as time, priority, severity of the situation and descriptions. WEF enables programmatic correlation technology such as Complex Events Processing.

2.6 Complex Events Processing

Complex Event Processing [15], or CEP, is primarily an event processing concept that processes events from an event cloud with the goal of identifying a specific sequence of events within the cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events.

2.6.1 Events and Complex Events

An event is an object that is a record of an activity in a system. The event driven programming model is applied in many technologies including Java Swing [24], Java Messaging Service (JMS) [24] and Microsoft Messaging Queue Server [17]. A Complex Event is an event that is an abstraction or aggregation of other events [15]. In other words, it is an event that occurs because other events happened.

For example, when planning a trip, one can use a travel web site to book flights and hotels as well as to rent a car. They are available on the web site because a number of other events took place — room events from hotels showing vacant rooms, ticket events from the airlines showing plan schedules, car events from rental companies showing available cars and so on. When flights or rooms are unavailable, more information, such as bad weather reports, gasoline price increases, or some other factors leading up to the desired event, may be provided by way of explanation.

Events are related in various ways, including the following [15]:

- Time. Time is a relationship that orders events. For example, event A happens before event B. Typically, when an event's activity happens, the event is created and it is given a timestamp.
- Cause. If the activity signified by event A has to happen in order for the activity signified by event B to happen, then A causes B.
- Aggregation. If event A signifies an activity that consists of the activities of a set of events, B1, B2, B3… Bn then A is an aggregation of all the Bi's Conversely, the Bi's are members of A.

This thesis focuses on the time-based relationship because events in WSDM have explicit timestamps indicating when they are generated.

2.6.2 Event Patterns

An event pattern is a template that matches certain sets of events [15] which indicate situations or system problems. In order to react to situations or diagnose problems automatically, the first step is to detect patterns in events.

One way to describe patterns is to write the pattern in an event pattern language [15]. Pattern matching has existed for a long time. For example, Google provides a user interface for users to type a sequence of key words and search for them in trillions of web pages to find matches. Some examples of patterns of interest in management are:

All database hit rate values collected in the last hour;

All web site visits that happened yesterday from 9 am to 5 pm;

All printers, started successfully today, which are short of paper.

The first pattern matches all hit rate events from the database that were collected during the last hour. The second pattern matches all HTTP calls that occurred during a specific time period. The third pattern is more complex, matching all printer events that indicate a successful start and later a lack of paper.

2.7 Autonomic Web Service Environment (AWSE)

A Web Service environment typically consists of a collection of components including HTTP servers, application servers, database servers, and Web Service applications [30]. In the AWSE architecture shown in Figure 2-3, each component is autonomic, that is, self-aware and capable of self configuration to maintain a specified level of performance. System-wide management of the Web Service environment is facilitated by a hierarchy of Autonomic Managers that query other managers at the lower level to acquire current and past performance statistics, consolidate the data from various sources, and use pre-defined policies and Service Level Agreements (SLA) to assist in system-wide tuning.

In Figure 2-3, the components on the left are Autonomic Elements. An autonomic element is a component augmented with self-managing capabilities. An autonomic element is capable of monitoring the performance of its component, or managed resource, (such as a DBMS or an HTTP server), analyzing its performance and, if required, proposing and implementing a plan for reconfiguration of the managed resource. Every component has two interfaces: the Performance Interface and the Goal Interface. The Performance Interface exposes methods to retrieve, query and update performance data, while the Goal Interface provides methods to query and establish the goals for an autonomic element.

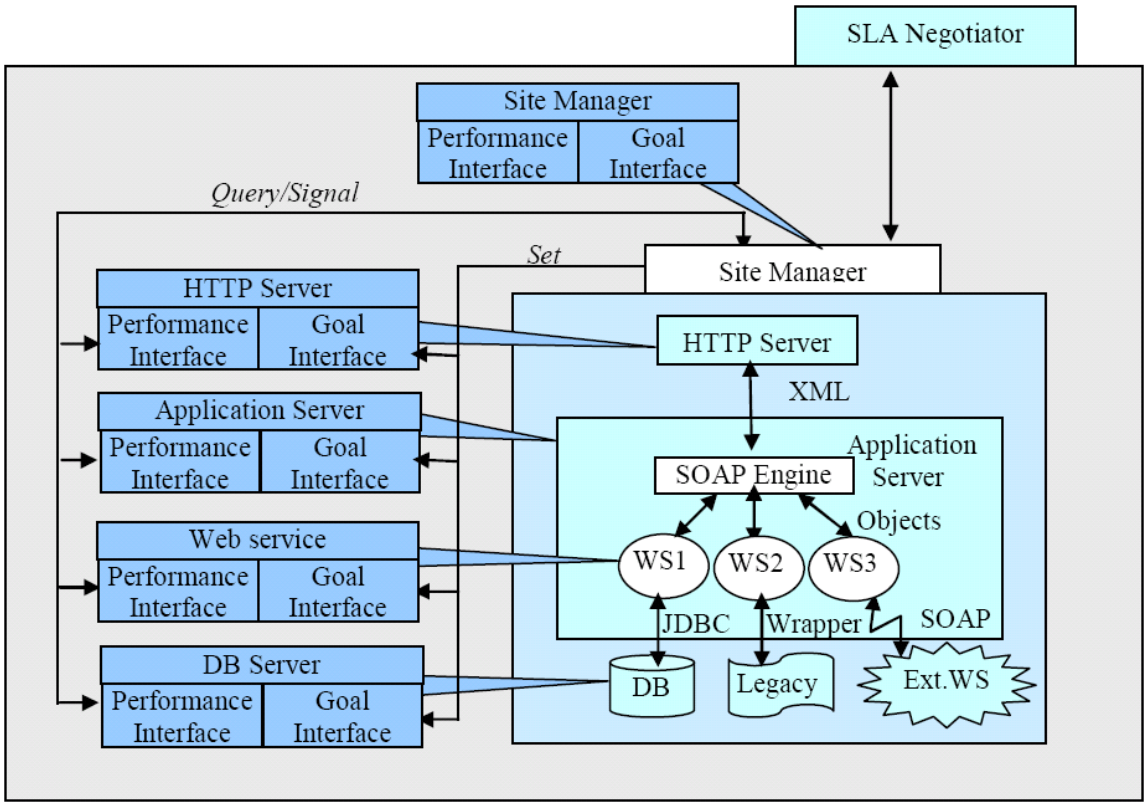

Figure 2-3 Autonomic Web Service Architecture (Reproduced with permission [30])

A Site is a collection of components and resources necessary for hosting one or more Web Service provided by an organization. A Site typically consists of HTTP servers, application servers, SOAP Engines, and Web Service. Web Service typically provides Web accessible interfaces or applications that can connect to other backend applications such as external Web Service or database management systems.

The Site Manager in Figure 2-3 is implemented as a Web Service. It exposes an interface through which the whole site can be accessed by other site managers or external components. This interface can be used by for SLA compliance monitoring. The Site Manager is responsible for monitoring the overall performance of the Web Service. It collects the performance data via the components' performance interfaces. The information required by the component for self-

management may differ from that required for overall system management by managers at the site level. For instance, a DBMS focuses on low level resources such as I/O and CPU usage to maximize performance. This information is available through the components management interface.

AWSE provides a general framework for WSMS to fit in between site manager and autonomic elements. AWSE defines one of the goals of WSMS: collect performance data from monitored systems and send information to manager. WSMS helps AWSE detect problems while AWSE takes actions to react to solve problems automatically.

2.8 Related Work

There has been a lot of work dealing with monitoring. Sahai and et al introduces an Automated Web Service Service Level Agreement monitoring engine [22] to collect performance data, model the data, and evaluate the SLA at certain times, or when certain events happen. This engine keeps track of message exchanges among Web Service by overwriting the Apache SOAP toolkit. It correlates individual messages by attaching a Global Flow (GF) ID to the messages. One limitation is that the GF ID is not a standard in Web Service, so the monitoring functionality is restricted to its own engine. Another difference between this and our work is that the monitoring engine focuses on SLAs where our monitoring approach can be used for both SLAs and problem determination.

Keller [11] and Ludwig proposed a Web Service Level Agreement (WSLA) framework to specify and monitor SLAs for Web Service. The WSLA framework consists of a WSLA monitoring service, which includes a Measurement Service and a Condition Evaluation Service. The Measurement Service is designed to back up metric definitions by a rich set of functions. It supports multiple data sources which interpret measurement instructions and execute

corresponding actions to read measurement data. Measurement Services work as a Web Service entity to transfer metric values at runtime. The Condition Evaluation Service supports a wide range of predicates. It offers a Web Service interface to receive metric updates from the Measurement Services. The author also defines a WSLA language to help specify the condition and metrics. This language is specific to this implementation and is not a standard. Additionally, this language is designed for SLA management, and is not intended for general purpose use.

Pistore and et al. [20] devised a planning technique for the automated composition and automated monitoring of Web Service specified as Business Process Execution Language for Web Service (BPEL4WS) processes. The planning technique automatically generates a monitor of the process, which is a segment of code that can detect and signal whether the external partners behave consistently with the specified protocols. However, this technique changes the interface of an existing Web Service to provide an observable behavior. This may be not acceptable in some cases.

Zulkernine and et al [33] proposed a middleware solution to monitoring processes based on composite Web Service in a distributed environment. In this paper, the authors described a Performance Monitor framework to monitor SLA compliance as well as a Web Service reputation knowledge base. This approach is a lightweight solution on the service consumer side to reduce the monitoring overhead especially when clients have only limited resources. In order to do so, it defined a XML schema to carry the monitoring information, which is not a Web Service standard.

Renesse and et al [21] developed a distributed information management system called Astrolabe. It monitors the dynamically changing state of a collection of distributed resources, reporting summaries of this information to the managers. It keeps computing summaries of the data it collects using on-the-fly aggregation. However, it uses a peer-to-peer protocol and a restricted

form of mobile code based on Standard Query Language (SQL). Therefore, the implementation may not be applicable for monitoring some legacy systems.

Comparing our monitoring system to related work, the novel features incorporated in our system are: 1) monitoring Web Service without changing the monitored systems; 2) employing standards such as WSDM and XQuery [1] which is well known to end-users and relatively easier to use; 3) correlating events by using the elements defined in Web Service specifications.

Chapter 3

Web Service Monitoring System Design and Functionality

Our monitoring system, WSMS observes events happening in a WSDM environment and reacts to the situations represented by those events. This chapter presents the design of WSMS beginning with the architecture. Then we describe the detailed design of a Sensor's four components as well as how Sensors can be combined into Sensor hierarchies. The specification of policy and the policy manager are also discussed. Finally, we present details of events, complex events, and outline how policy is distributed.

3.1 Architecture Overview

This section presents an overview of the system architecture, identifies the functional and non-functional requirements of the Sensor system, and describes a scenario including various use cases. The WSMS architecture is depicted in Figure 3-1.

This architecture shows the three main components of WSMS: Sensor, Policy Manager, and Sensor Manager. Before we discuss these three components, we need to introduce a concept applied here: policy-based design.

Policy: A policy is a set of considerations designed to guide decisions of courses of action [16]. Here are some policy examples:

1. The database must back up nightly between 2 am and 3 am.
2. Only management and the Human Resources department have the access to employee's personal records.
3. The intranet firewall only allows Telnet and HTTP to traverse it.

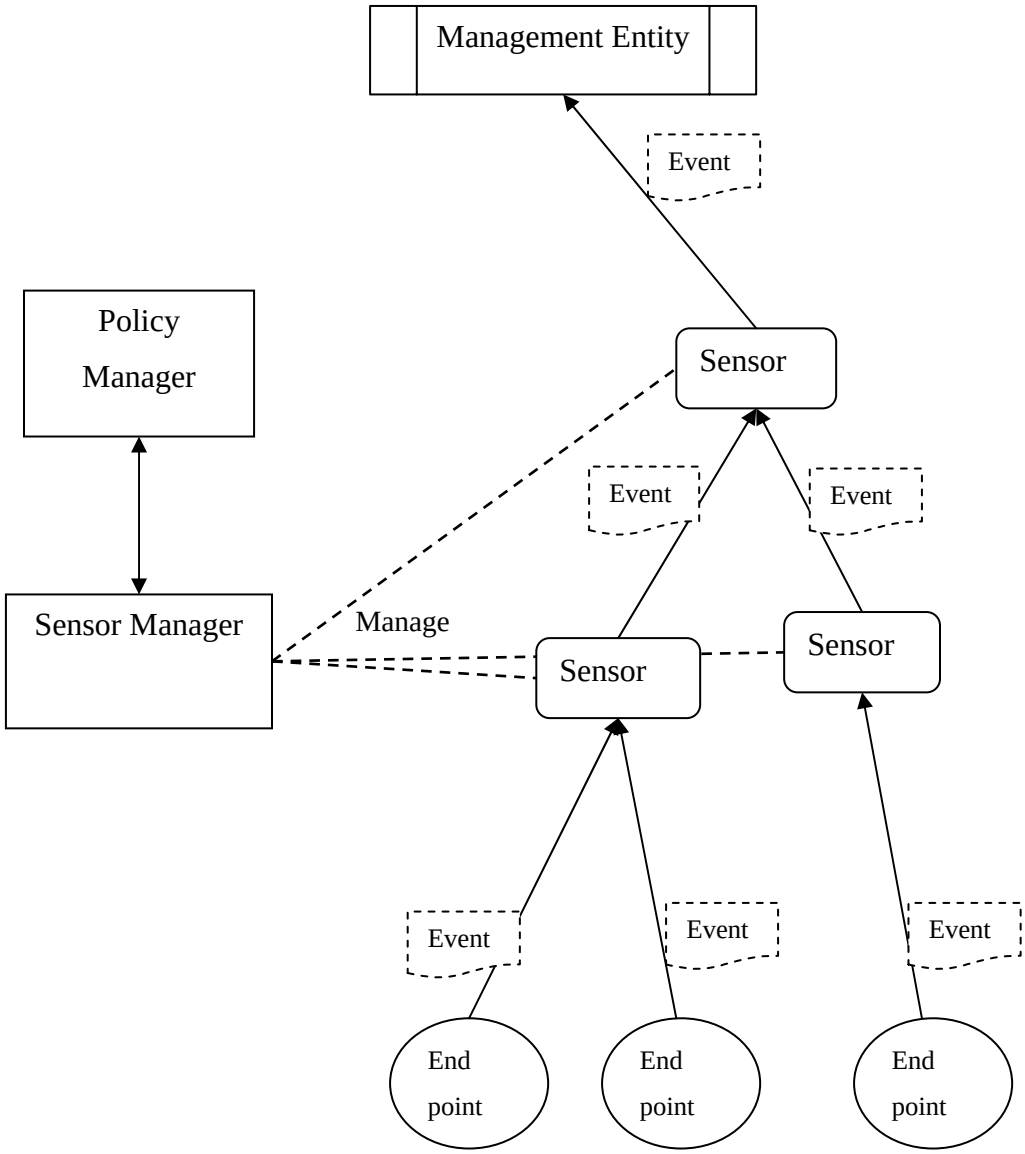

Figure 3-1 WSMS Architecture

The above policies regulate the actions of some roles and systems in an IT environment. We use policy to define the behaviors of the Sensors. A Sensor Manager manages the Sensors through the policies. Therefore, a Sensor Manager and its managed Sensors can reside on different machines and communicate with each other via Web Service calls. We discuss this further in Section 3.5.

Sensor: A Sensor is a light-weight component that consumes WEF events, processes them in real time, and generates and publishes complex WEF events that are consumed by other Sensors. A Sensor has its own life cycle, which is managed by a Sensor Manager. A Sensor can be deployed dynamically at runtime, or uninstalled when necessary. Sensors may consume events produced by other Sensors or from Web Service endpoints based on topics defined for managed resources and producer Uniform Resource Identifiers (URI) in a Sensor's policy.

Events are saved into a repository unique to each Sensor. When a new event arrives, the Sensor runs queries on the events history using the rules, or policies, that it holds, looking for matches. Once a match has been found, the Sensor publishes events to specific topics to which other Sensors subscribe. For instance, the top level Sensor in Figure 3-1 publishes events to a management entity that can be a user or a program. These events provide suggestions or information to the management entity upon which it can base its decisions.

Consider, for example, a Sensor that is monitoring an HTTP server. A Sensor subscribes to the Web Service URI representing the server, "http://localhost/httpserver/services/httpserver", and the topic, "ResponseTime". Therefore, the Sensor receives messages regarding response time changes from the HTTP server and saves these messages into a repository. When a new event is received, the Sensor uses past events as well as the current event to determine if there has been an increase in the response time. If so, the Sensor publishes a complex event indicating that "the HTTP Server is slow". This event is received by higher level Sensors that have subscribed to receive such events.

Sensor Manager: A Sensor Manager is responsible for managing and monitoring a cluster of Sensors. A Sensor Manager communicates with the Policy Manager to retrieve its policies before the installation of new Sensors or prior to updating existing Sensors.

Policy Manager: The Policy Manager is responsible for managing policies. We assume that a policy can be created or updated by some external source. When the Policy Manager receives a new policy, it sends out a notification to the Sensor Managers indicating that a new policy is available, along with instructions as to how to obtain the policy.

With these loosely coupled components, WSMS monitors Web Service by analyzing events within the network without interfering with the monitored Web Service endpoints.

3.2 Scenario

To illustrate how WSMS would work, we present the following scenario of Web Service monitoring shown in Figure 3-2. Assume that two components are monitored, namely a DBMS and a Web Service. Each component has multiple properties that indicate the performance, such as hit rate, throughput, rejection rate and the Web Service's call mix, which is a metric used to indicate the percentage of each type of call to the Web Service.

AWSEdb and AWSEws are two WSDM endpoints to collect the metrics related to the DBMS and the Web Service, respectively. They check the status of the components and send out metric values (hit rate, throughput, etc) if there has been some changes to these components. For example, the DBMS sends an event to AWSEdb when the buffer pool hit rate falls. The components send messages to four topics representing four metrics respectively, as shown in Figure 3-2. Therefore, the events reflect the components' performance.

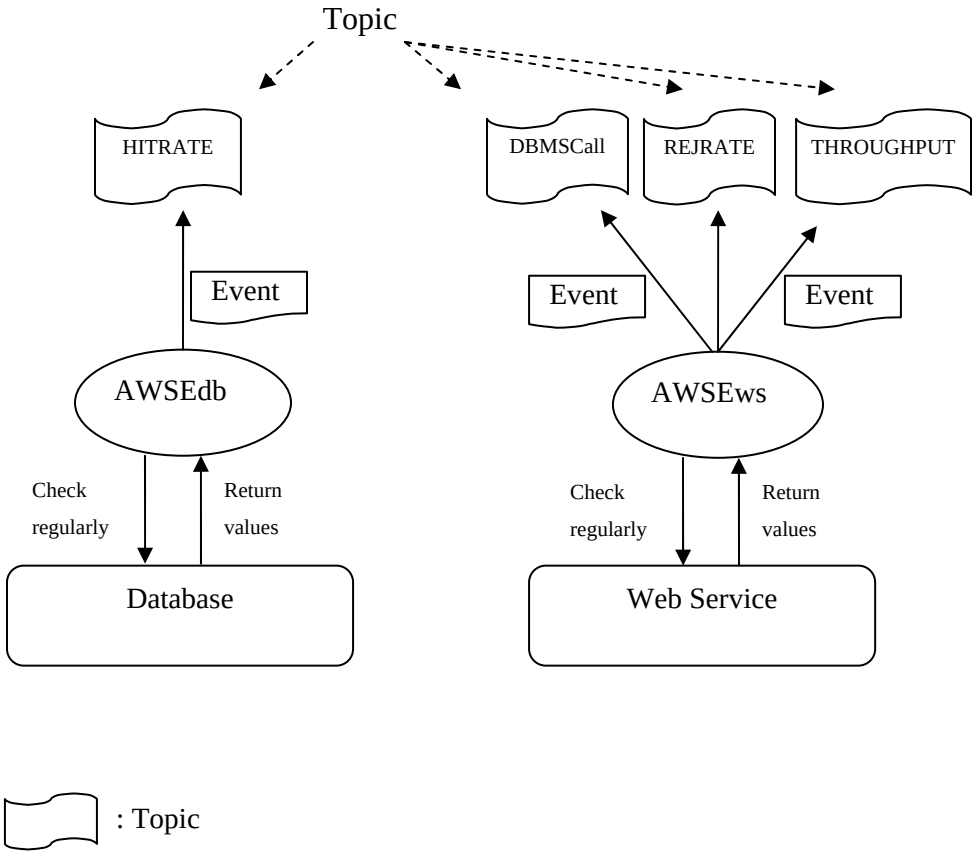

Figure 3-2 Send Performance Metrics to Topics

There are four types of events:

1. DBMS Hit Rate. This event indicates a change in the hit rate of the DBMS buffer pool. The hit rate indicates the probability that a requested page is found in memory. A higher hit rate indicates better performance. This metric is published by the AWSEdb endpoint.
2. Web Service Call Mix. The Web Service has two types of calls which use the DBMS differently. We refer to these calls as Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). The Web Service Call Mix events are

generated when the percentage of calls to the OLAP method changes. For example, if the workload consists of 25 percent OLAP calls and 75 percent OLTP calls, the Web Service Call Mix is 25. If the workload changes and the number of OLAP calls increases say to 50 percent, then a Web Service Call Mix event will be generated. These events are published by the AWSEws endpoint and provide an indication of a workload change which, in this case, changes the way in which the buffer pool is used. Therefore, this information, in combination with a falling hit rate, indicates that the problem may be solved by increasing the size of the DBMS buffer pool.

3. Throughput. A throughput event is the average number of transactions processed by the Web Service per second. A higher value indicates better performance.
4. Rejection Rate. Rejection rate events indicate the number of calls to the Web Service that cannot be processed due to lack of available connections to the DBMS. A low value for the rejection rate indicates better performance.

3.3 Sensor

A Sensor has four components: Events Collector, Events Analyzer, Events Repository and Events Generator. Figure 3-3 shows the architecture of a Sensor.

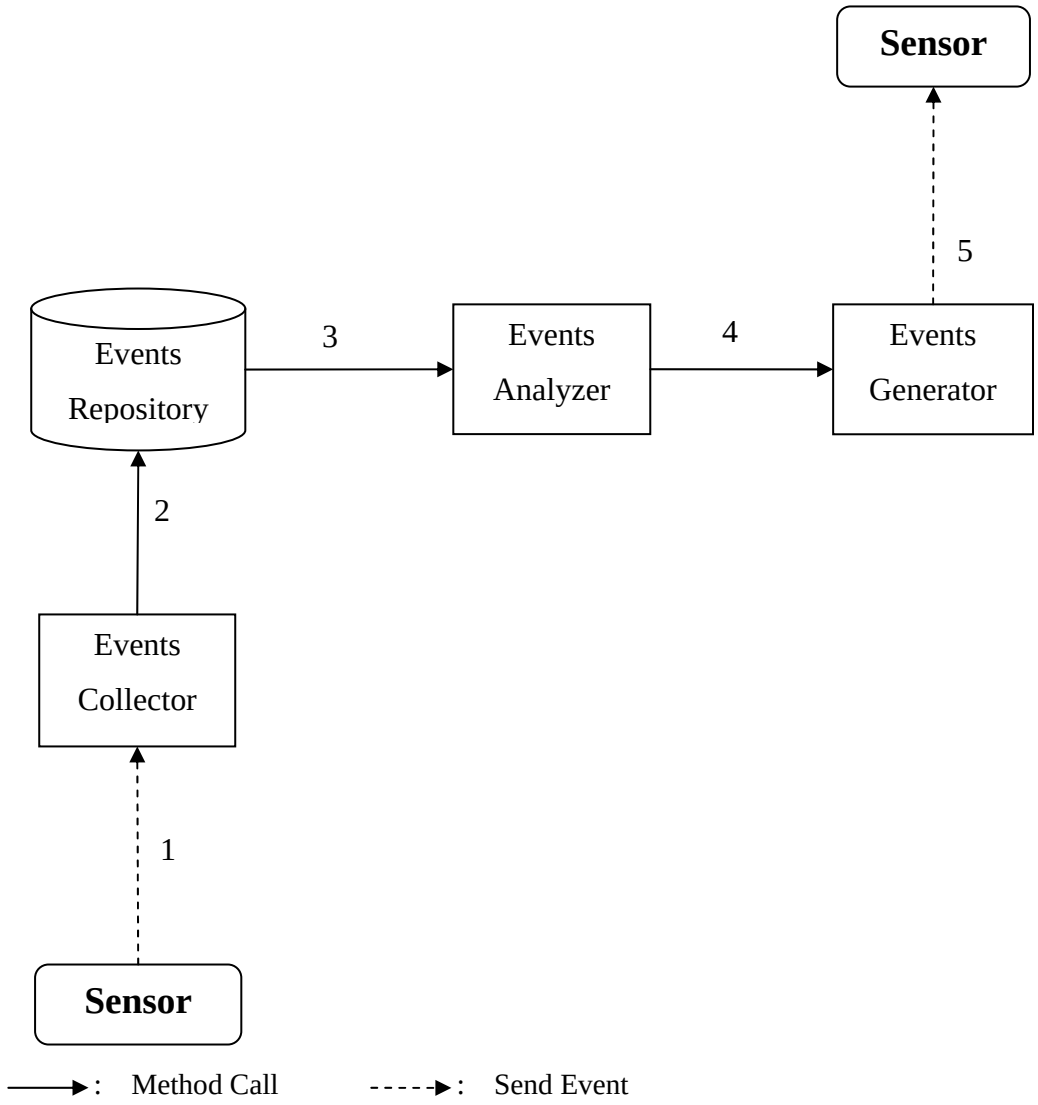

Figure 3-3 Sensor Architecture

Each Sensor is an independent entity with publish/subscribe messaging capabilities to communicate with other Sensors. A Sensor can subscribe to multiple topics and it can publish complex events to a topic which is consumed by another Sensor. The relationship, therefore, between Sensors is many-to-many. In Figure 3-3, there are five message flows and calls:

1. The Events Collector consumes events from one or more Sensors or devices.

2. The Events Collector saves the events into repository for further analysis.
3. When a new event is saved, the Events Analyzer is called by Events Collector to run queries to find matches.
4. Once the Events Analyzer finds a match, it passes references to the sequence of events which result in the match to the Events Generator.
5. The Events Generator publishes complex events to the other Sensors.

3.3.1 Event Collector

The Events Collector accepts incoming events and stores them in an events repository for future analysis. Figure 3-4 shows how event consumers and producers work together to exchange events. In this figure, there are four producers to publish events to three topics. On the other side, event consumers subscribe to these topics. The events are pushed to the consumers once they are published. Therefore, events are transferred from producers to consumers. For example, in Figure 3-4, producer #3 publishes one event to Topic B. Both Consumer #1 and Consumer #2 receive the same event because they subscribe to Topic B.

In a Sensor, the Events Collector works as an event consumer. The topic name it subscribes to is specified in the Sensor's policy. An Event Collector may subscribe to multiple Topics.

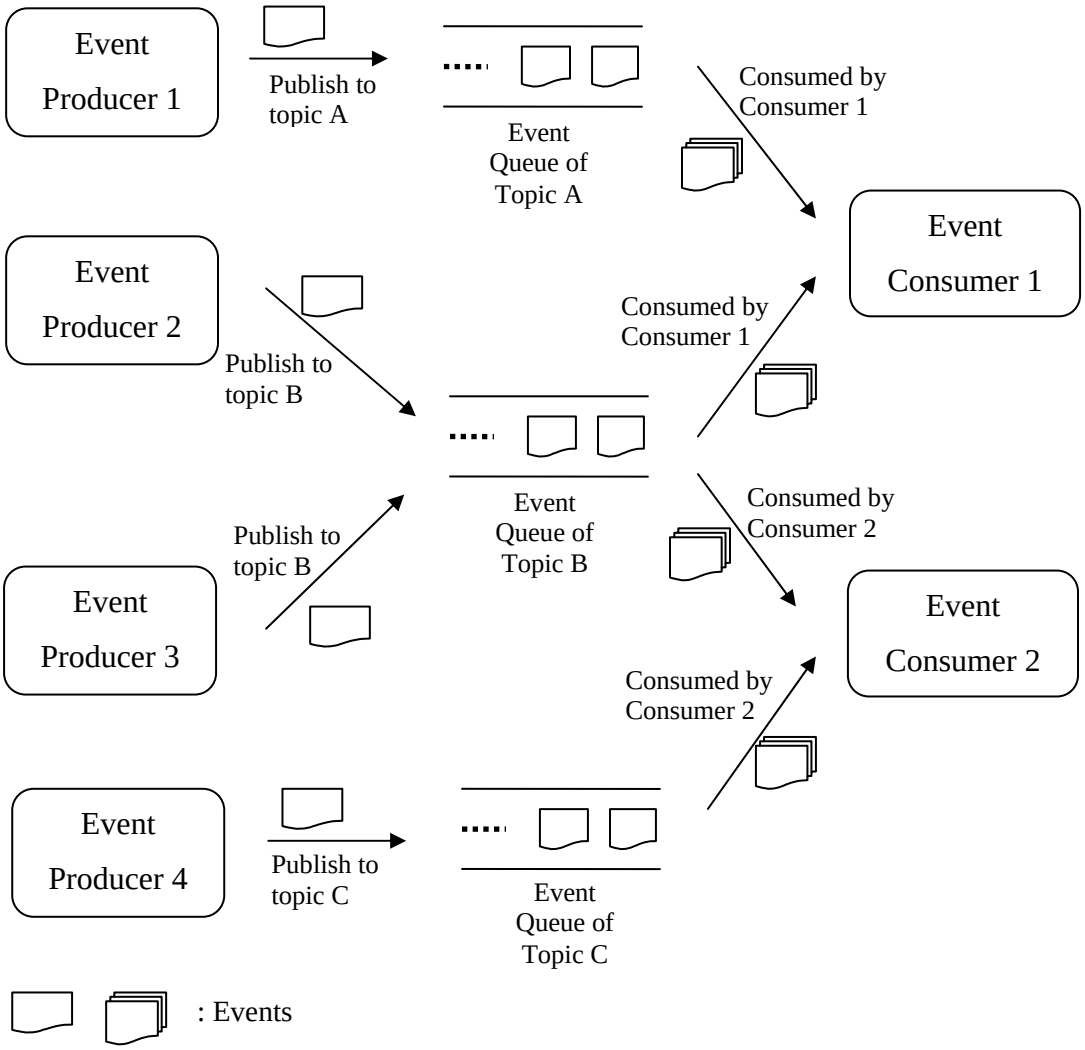

Figure 3-4 Events Exchange Between Producers and Consumers

3.3.2 Events Repository

The Events Repository provides storage to save events and an interface through which events are queried. Events in our case are in XML format. Maintaining the XML structure of the events message reduces the complexity of Sensors since no parsing is required for storage and a standard XML query language such as XQuery can be used to query the repository.

3.3.3 Events Analyzer

The Events Analyzer is responsible for events processing. It submits the events pattern, as specified in the policy, to the Events Repository and processes the results. If matches are found, it composes a complex event and passes it to the Events Generator to publish. The pattern and the content of the complex event are both specified in the Sensor's policy.

3.3.4 Events Generator

The Events Generator is responsible for publishing complex events that will be consumed by other Sensors or the decision-maker. Events are generated and published whenever the Events Analyzer detects a pattern that matches a Sensor's policy. As Figure 3-3 shows, the Events Generator plays the role of producer. Both the message body and topic it publishes are defined in the Sensor's policy.

3.4 Sensor Hierarchies

A Sensor is meant to be a simple entity with a view of only a small part of a functioning system. Each Sensor, as described above, compares incoming events against its policy to recognize particular patterns in the data. These simple Sensors can be arranged in hierarchies to perform complex analysis. One or more Sensors monitor parts of the system and feed events to upper level Sensors which in turn can accept events from multiple sources and look for more complex data patterns. Consider, for example, the scenario in Section 3.2. The two components send out four different performance metrics which are consumed by different Sensors. We can organize these Sensors to form a hierarchy to analyze the metrics and detect problems, as shown in Figure 3-5.

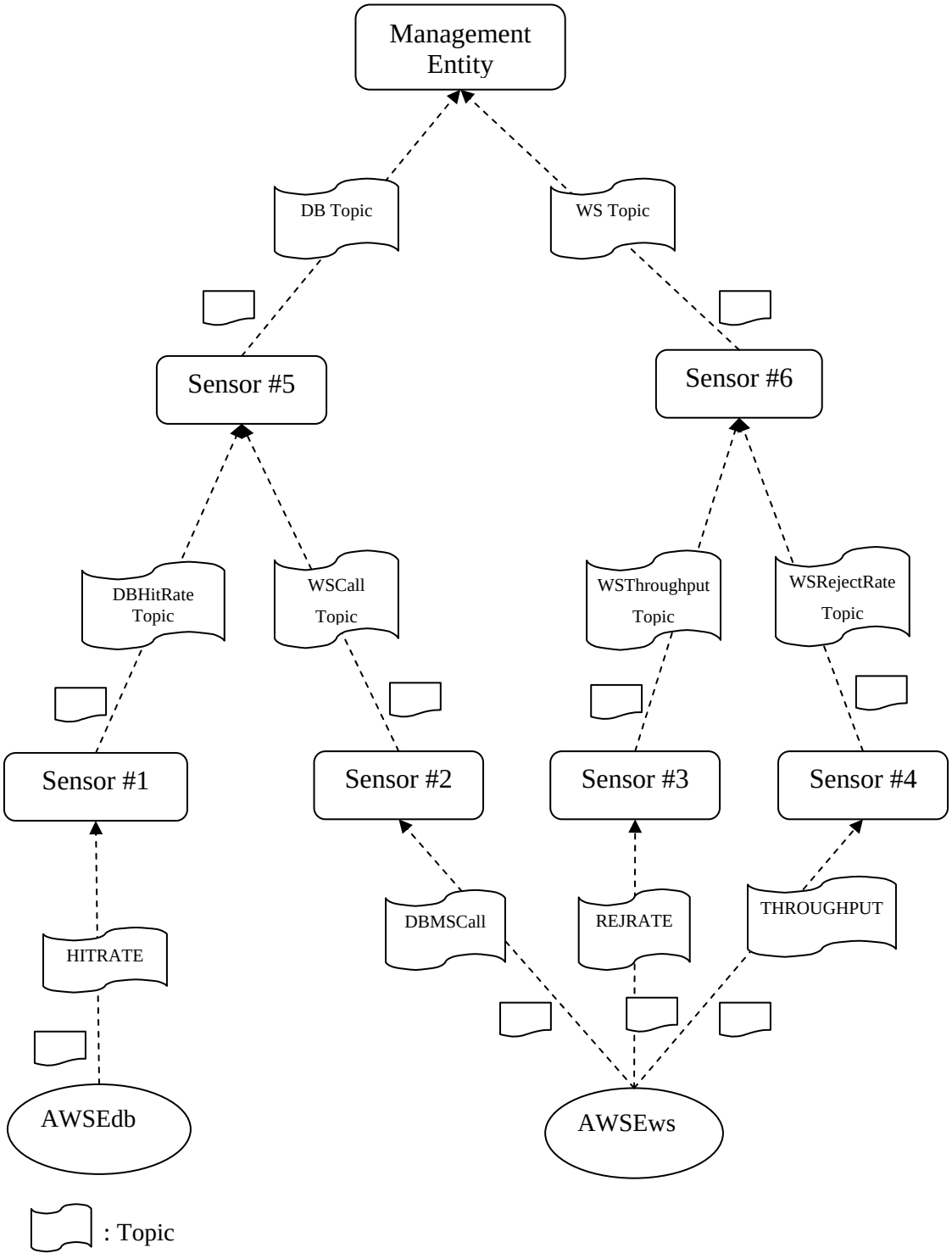

Figure 3-5 A Hierarchy of Sensors to Monitor Devices

1. The lowest level Sensors collect information from the managed resources. The Sensors keep events of interest, as defined by their policy, and filter out others. For example, the Sensor monitoring the database (Sensor #1) only keeps the events indicating exceptional situations such as a more than 50% increase in workload and discards events indicating normal performance. The Sensor's policy determines whether or not a complex event is generated. Typically a set of events are examined, either some number of events or, the set of events collected over a certain time window. If the pattern specified in the policy is detected, a complex event is generated and published for consumption by other Sensors.
2. The Sensors in the intermediate levels consume events sent from the Sensors in the level immediately below and publish complex events to the Sensors in the level immediately above.
3. The top level management entity consumes events from lower level Sensors and takes corresponding reactions.

At the lowest level of the hierarchy, each Sensor has a very narrow view of the overall system. They collect events from a very small portion of the system. As we move up the hierarchy, the picture becomes broader. Upper level Sensors may receive information from various sources providing a broader understanding of overall system performance.

A Sensor Manager manages a hierarchy, or domain, of Sensors that cooperate to process events. For example, in Figure 3-6, the Sensors are divided into two domains, Domain A and Domain B, both of which are circled by dotted lines.

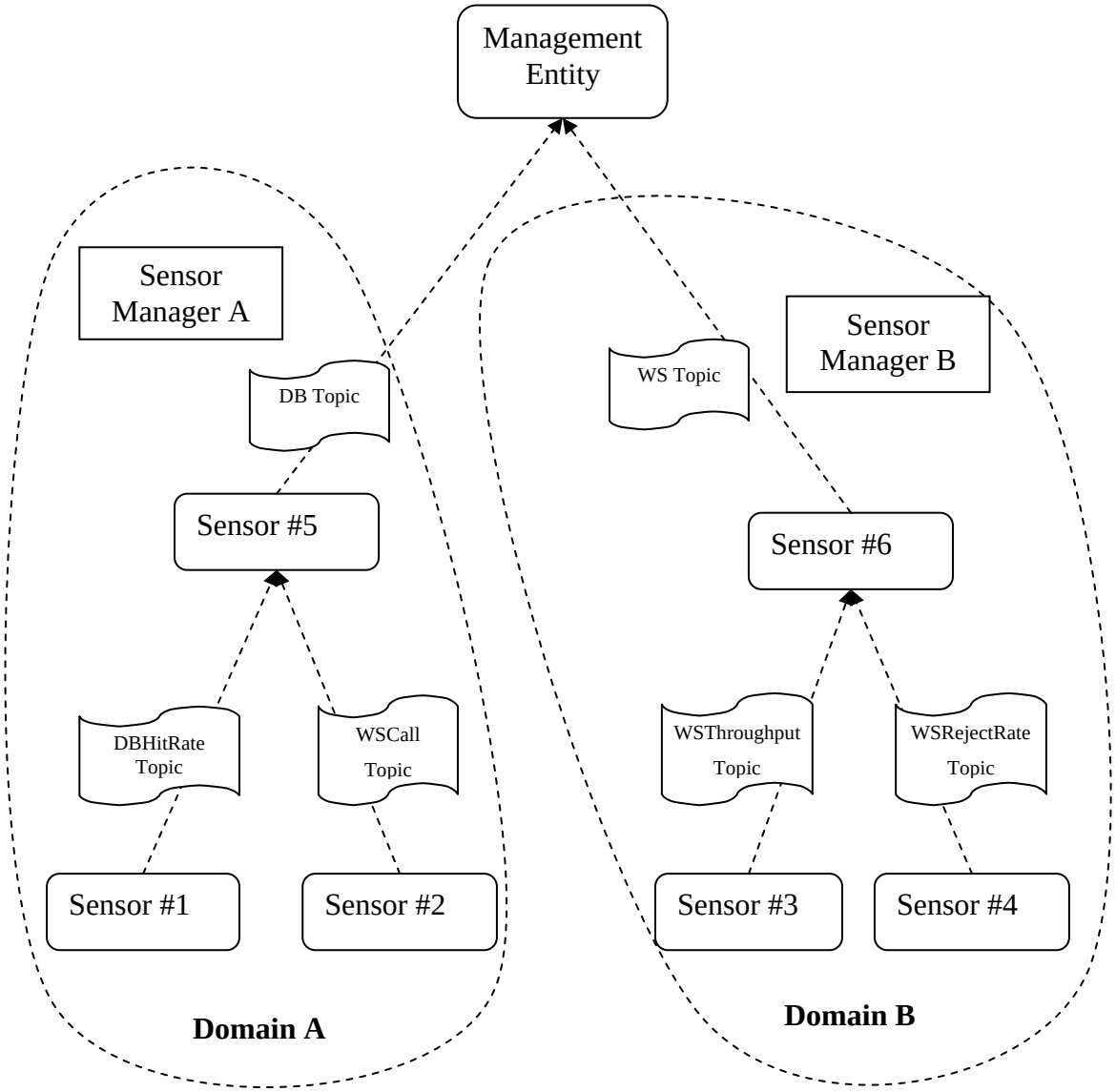

Figure 3-6 Two Domains of Sensors

In this figure, there are two domains with three Sensors respectively. They can be partitioned logically. For example, the Sensors in one domain monitor the DBMS while the Sensors in the other monitor the Web Service. Alternatively they can be partitioned physically. For instance, one domain is put on the intranet for a local site and another one is put on the internet for a remote site.

A Sensor Manager coordinates the interactions between the Sensors and the Policy Manager. It communicates with the Policy Manager to retrieve policies, interpret the policies, extract data from the policies, generate Sensors based on the policies, and deploy the Sensors. Additionally, it manages the life cycle of Sensors such as initialization, installation and termination.

3.5 Policy

Autonomic Computing supports the design of a policy-based distributed management architecture, which is used to translate high level policies into low level system actions. This architecture is popular because it contributes semi-automation to the system and reduces the responsibilities of the administrators. This concept is applied in the design of WSMS. We abstract Sensor's behavior to design a policy for it. Each Sensor entity, consumes events from one or multiple sources, performs pattern matching against the events, and outputs events. Therefore, we define a schema of the policy as represented in class diagrams in Figure 3-7.

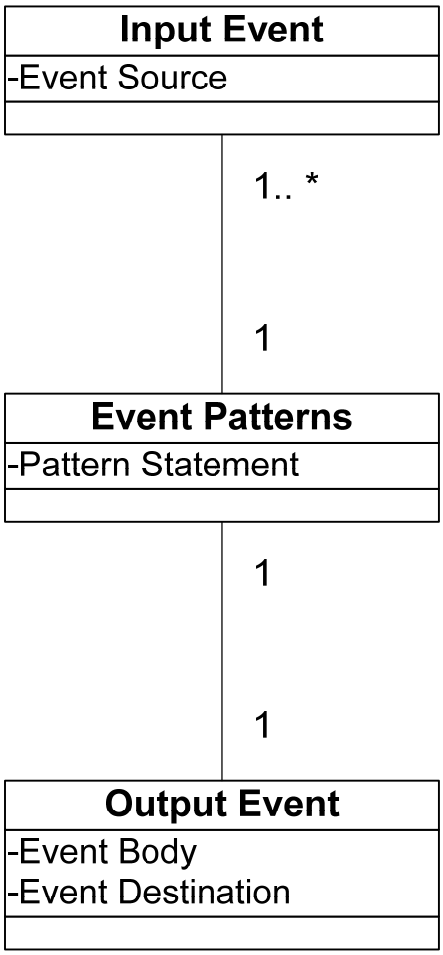

Figure 3-7 WSMS Policy Schema

Input Event has one attribute, Event Source, to identify the source of the event. Each Sensor may consume events from multiple sources. This attribute is registered in a Sensor so that this Sensor is able to consume all events from that source.

The second part of the policy is the Event Pattern which defines the policy for the pattern matches. In our design, every Sensor has one pattern matching policy to reduce the complexity and make the Sensor as lightweight as possible. Therefore, the relationship between Input Event and Event Patterns is many-to-one.

The third part in a Sensor policy is the Output Event which defines the output event (Event Body) and their destination (Event Destination), that is, the address of the event destination. There is one and only one output type for each Sensor, so the relationship between Output Event Types and Event Patterns is one-to-one. Policies for Sensors in the same domain compose a domain policy list. A Sensor Manager interprets this list and deploys a group of Sensors.

Consider our previous example from Figure 3-5. There are two domains consisting of six Sensors in this hierarchy, thus requiring two policy lists and six policies in total. For example, Sensor #1 takes events from the Database as input, looks for patterns indicating a hit rate decline, and outputs events to Sensor #5. The policy for Sensor #1 looks like the following:

<Input_Events>

<Events_Source>reference to Database</Events_Source>

</Input_Events_Type>

<Event_Patterns>

<Pattern_Statement>patterns used to find matches</Pattern_Statement>

</Event_Patterns>

<Output_Events>

<Event_Body>"Hit rate is declining" </Event_Body>

<Event_Destination> reference to Sensor #5</Event_Destination>

</Output_Events>

Both of Sensor #5 and Sensor #6 send events to management entity or manager. The policy for Sensor #5 looks like the following:

<Input_Events>

<Events_Source>reference to Sensor #1</Events_Source>

<Events_Source>reference to Sensor #2</Events_Source>

</Input_Events>

<Event_Patterns>

<Pattern_Statement>patterns used to find matches</Pattern_Statement>

</Event_Patterns>

<Output_Events>

<Event_Body>"More resources are needed"</Event_Body>

<Event_Destination>reference to manager</Event_Destination>

</Output_Events>

If all of the six Sensors are in one domain, the policies are put into one policy list.

3.5.1 Distributing Policies

A publish/subscribe methodology minimizes the coupling degree between components. Not only does it improve the reusability of components but also reduces the complexity of maintenance. Therefore, this method is used for the distribution of policies to the Sensor Manager(s). By doing this, the management of WSMS is simplified. We change the behavior of Sensors in WSMS by writing a new policy in XML format and publishing it to the appropriate Sensor Managers. WSMS installs the policies automatically. In other words, we manage WSMS by managing policy, which is simpler than managing the Sensors directly.

The Policy Manager receives a list of policies from some external source then transfers it to the Sensor Managers using a double handshake methodology. Policy Manager first sends an event to inform the Sensor Managers of the new policies. The relevant Sensor Managers then retrieve the policies.

3.6 Events Pattern Language

Event Pattern Language (EPL) is a set of computer languages in which we can precisely describe patterns of events [15]. WSMS requires an EPL implementation for detecting event patterns. As we discuss in chapter 2, there are three types of correlation between events, namely time, causality and aggregation. To support these correlations Luckham lists the operators needed in an EPL [15] implementation, as shown in table 3-1.

According to the schema of a WSDM Event Format, the relationship of events is time-based because of the timestamp within message bodies. We therefore consider the operators *P<Q*, *P and Q*, *P or Q*, *P not Q* and *P U Q*. The other operators *P→Q*, *P||Q* and *P~Q* in the table are only needed when the relationships are cause-based or aggregation-based.

XQuery is chosen as the EPL implementation in our case. It supports the operators in Table 3-1 indirectly by its embedded functions:

- Logical operators. They can be implemented by guards in the Where clause.
- Set operators. XQuery has some keywords such as UNION, INTERSECT, and EXCEPT to aggregate tuples.
- Structural operators, *P<Q* can be implemented by guards in the Where clause and joins in XQuery.

Table 3-1 EPL Operators in CEP

Operator	Name	Description
Structural operators		
P → Q	Causes	All events in the match of Q are caused by all events matching P
P || Q	independent	Any Event in the match of P is independent of every event in the match of Q, and conversely.
P < Q	Before	Any event in the match for P has an earlier timestamp than all events in the match for Q
Logic operators		
P and Q	and	The events in the match for P must match Q
P or Q	or	The events can match P or match Q
P not Q	not	The events in the match for P must not match Q
Set operators		
P U Q	union	Two sets of events, one matches P, another matches Q
P ~ Q	disjoint union	Two disjoint sets: one matches P, another matches Q

An example of pattern written in XQuery is as follows

Q: Identify a hit rate declining of 10 percentages.

A: xquery

declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";

declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ;

for $a in db2-fn:xmlcolumn('EVENT')/muws1:ManagementEvent,

$b in db2-fn:xmlcolumn('EVENT')/muws1:ManagementEvent

where xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime))

and fn:number($a/muws2:Situation/muws2:Message)-fn:number($b/muws2:Situation/muws2:Message)>10

return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>

Not only does XQuery meet our EPL requirements, but it also has the flexibility to extend in the future to support other events relationship such as causality and aggregation.

3.7 Summary

In this chapter, we introduce the architecture and detailed design of WSMS. In summary, WSMS includes a hierarchy of Sensors, several Sensor Managers to manage them, and a Policy Manager to distribute policy. The Sensors consume, process events, and produce complex events to a management entity.

Chapter 4
System Implementation and Testing

This chapter presents a proof of concept prototype of our WSMS design. We begin with the detailed implementation followed by a description of the development environment and tools used, including Eclipse TPTP [29], Apache Muse [26], IBM DB2 Database [10] etc. We continue with a discussion of the prototype with a testing scenario along with three cases to identify the capabilities of WSMS.

4.1 Implementation

This section describes the implementation of WSMS, starting with an overview. We then discuss the detailed implementation of Policy Manager, Sensor Manager and Sensor. We also discuss the relations and messages flow among them.

4.1.1 Overview

A Web Service endpoint is a referenceable entity, processor, or resource where Web Service messages can be targeted [2]. A WSDM endpoint is an endpoint with manageability capabilities defined in WSDM, such as resource identity, metrics, configuration and relationships, which can be composed to express the capability of the management instrumentation. It also has the capabilities defined in WSN [6], including notification producer and consumer.

An End Point Reference (EPR) is a combination of Web Service elements that define the address for a resource in a SOAP header [7]. An EPR consists of a URI, message reference parameters and data concerning the interface to be used. Basically, an EPR holds information to call a

service. The simplest EPR is usually a Uniform Resource Locator (URL) but it can also be much more complex.

WSDM endpoints exchange events by a subscription method. Every event producer has an EPR to which it refers. It also defines a topic which in most cases is related to the events it produces. Then the producer publishes events to this topic without knowing who consume them. On the other side, an event consumer subscribes to the topic defined by producer as well as the producer's EPR, namely Producer URI, in order to receive all of the events published to the topic by the producer.

In our case, we set up Sensor Manager and Policy Manager as two WSDM endpoints. Additionally, we define Sensor Manager as an event producer and Policy Manager as both a producer and a consumer. We will explain this later in the following section. In a double hand shake method, Policy Manager delivers policy to Sensor Manager which installs Sensors as Java instances at runtime. After Sensors are installed, they form a hierarchy to monitor systems.

Every Sensor has its own event input and output as we discussed in chapter 3. From the perspective of implementation, each Sensor has different topics to subscribe to and publish events to. Conceptually, they are independent WSDM endpoints to exchange events in the subscription method. However, in practice, we use Sensor Manager as the proxy to publish and consume events for all of the Sensors in one domain. All of the Sensors plus their Sensor Manager are therefore one WSDM endpoint. There are some advantages to this approach. First of all, every Sensor is a Java instance instead of a Java Web application package which is a must for being a WSDM endpoint. It is much easier to manage the life cycle of a Java instance which is only technically a thread rather than a web application. Moreover, keeping all Sensors as one WSDM endpoint makes our implementation lightweight to deploy. For example, in our scenario we may

have about ten web applications if every Sensor is a WSDM endpoint. Now we have only two. Finally, we improve the performance by reducing the number of Java Web applications since typically fewer Web application packages means better performance.

4.1.2 Implementing the Policy Manager

Policy Manager is responsible for not only for notifications that new policies are available, but also for holding policies for Sensor Managers to retrieve. Figure 4-1 shows how Policy Manager distributes policy to multiple Sensor Managers. The cloud in the figure is a virtual broker to hold all events that Policy Manager produces. The Sensor Managers subscribe to topics and receive appropriate events from the broker. The broker is implemented as a part of Apache Muse.

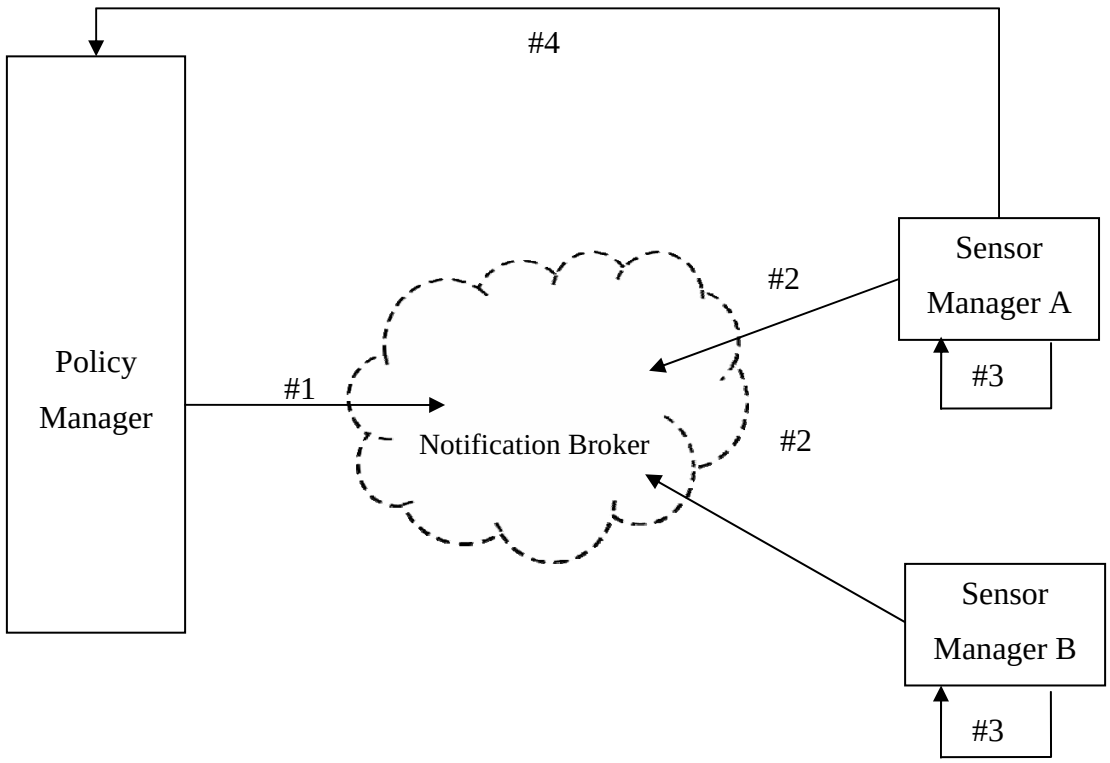

Figure 4-1 Double Hand Shake Policy Distribution

There are 3 steps for a policy distribution as follows:

1. Policy Manager sends an event. Once Policy Manager receives new policy lists, of which each is for one domain, it sends an event to a topic, indicating that new policy lists are available. It also contains the URL from which the policy can be downloaded.
2. Sensor Managers receive the events. There may be multiple Sensor Managers subscribing to the topic mentioned in step 1. All subscribed managers receive the events.
3. Sensor Managers retrieve policy. If the Sensor Manager decides to retrieve the policy list, it follows the instructions indicated in the event sent by Policy Manager to fetch the policy. After a Sensor Manager gets a policy list, it breaks the list into independent policies, builds a group of corresponding Sensors based on the policies, and installs them.

Policy Manager is implemented as a WSDM endpoint with the events producer capability. It defines a topic named "POLICY_NOTIFICATION". It also defines the EPR, http://localhost:8080/PolicyManager/services/PolicyManager, which is the policy producer URI. Having these two properties, it is able to publish events to distribute policy to Sensor Managers.

For example, when a new policy arrives at Policy Manager, it exposes the policy as a downloadable resource via HTTP and creates an event containing the URL of the policy. Then it sends this event to the topic POLICY_NOTIFICATION. Sensor Manager receives the event and downloads the policy. When there are multiple Sensor Managers, they subscribe to the topic POLICY_NOTIFICATION so that all of them are able to download the policy. It is Sensor Manager's responsibility to judge if the policy is related with the Sensors within its domain. If it is, it downloads the policy. Otherwise, it does not.

4.1.3 Implementing the Sensor Manager

Sensor Manager is responsible for ensuring the Sensors' behavior follows the policies. It is implemented as a WSDM endpoint with the capability of event consuming. It subscribes to the topic defined by Policy Manager, POLICY_NOTIFICATION in our case, as well as the policy producer URI defined above. When Policy Manager sends an event saying that a new policy list is available, The Sensor Manager will receive it and download the policy list via the HTTP URL embedded in the event. The following diagram shows this process. After the Sensor Manager gets the policies, it creates multiple Java instances of a Sensor, such as DBSensor and WSSensor in Figure 4-2, to implement the policies. The Sensor Manager manages the life cycle of these Sensors.

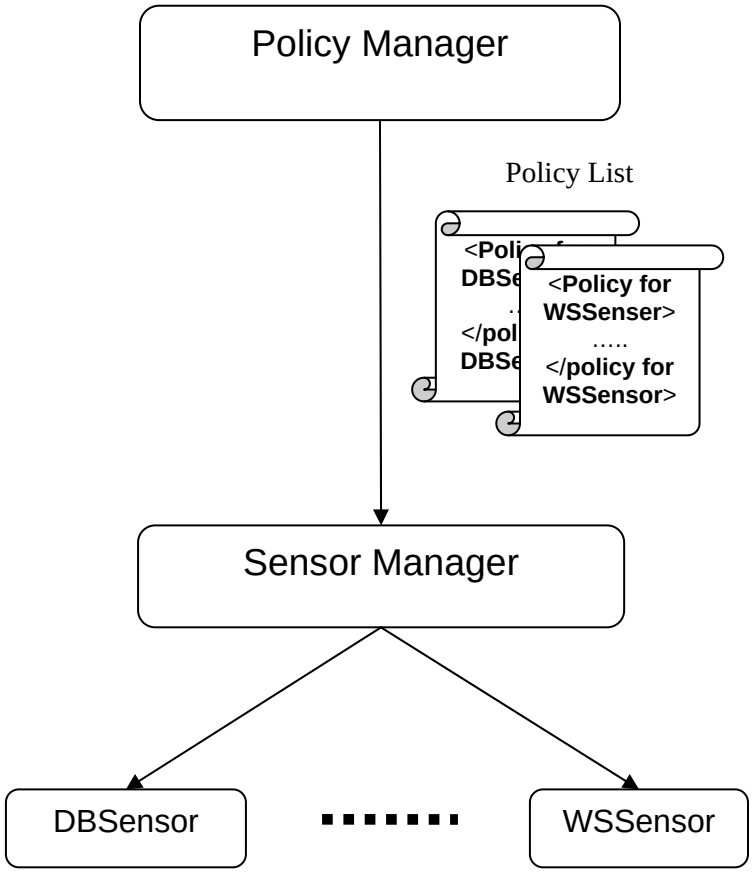

Figure 4-2 Sensor Manager Interprets Policy and Creates Sensors

A Sensor Manager also has the events consumer capability because it acts as a proxy for the Sensors in the same domain to publish and consume their events. As with the Policy Manager, Sensor Manager has its own EPR as a producer URI, namely http://localhost:8080/SensorManager/services/SensorManager. We can see that it is different with the URI of Policy Manager. Having this URI, combined with specific topics, Sensors can exchange events with each other. For example, suppose DBSensor sends an event to WSSensor which is in the same domain. At first DBSensor defines a topic, say “DB_TOPIC”. As we discussed, DBSensor shares the same producer URI with Sensor Manager. So WSSensor subscribes to the same topic and the producer URI mentioned above. Finally, DBSensor publishes an event to the topic and WSSensor can receive it.

4.1.4 Implementing the Sensors

Sensors are installed and initialized by Sensor Manager according to the policy list. As we discussed in chapter 3, every Sensor has four components: Events Collector, Events Analyzer, Events Repository and Events Generator. Sensors are conceptually WSDM endpoints except that they share the same producer URI with Sensor Manager. The sequence diagram of Sensors is shown in Appendix B.

Events Collector receives events by using the events consumer capability. This component is initialized by specifying a topic and producer URI of the event source to which the Sensor subscribes and so can receive events from that source through its Sensor Manager.

Events Repository is implemented as a combination of database tables and a Java class managing the access to them. Every Sensor has an independent table to store events and each event is stored as a row in the table. In our case, all of the Sensors share the same database and the table

structures are all the same except for the table names. For convenience, we use the Sensor's name as the table's name. The table structure is shown in Table 4-1.

Table 4-1 Table Structure of Events Repository

Field	Type	Null	Key
EventID	Varchar(20)	N	Y
Event	XML	N	N
EventTime	timestamp	N	N

There are three columns in the table: EventID, Event and EventTime. EventID is the unique id of the events. In practice, it is a hash string generated by the event producer which is embedded in the events. Events Repository component gets this string out of the event body and saves it in this column. It is also the primary key of this table. Event column stores the message body in XML format. It contains all of the elements for Events Analyzer to query. EventTime column saves the timestamp when events arrive. All of the three columns are not allowed to be null.

When a new event is saved into the repository, Event Analyzer is called. Through Java Database Connectivity (JDBC), this component can submit statements written in XQuery to the repository and get results to judge if matches are found. The XQuery statement is specified in the Sensor's policy.

Event Generator uses the events producer capability. It defines a topic and publishes an event to this topic whenever Event Analyzer finds a match. It shares the same producer URI used by Sensor Manager.

Figure 4-3 shows a class diagram of a Sensor's components. In the diagram we can tell that Events Collector uses the events consumer capability to collect events while Events Generator uses the events producer capability to publish events.

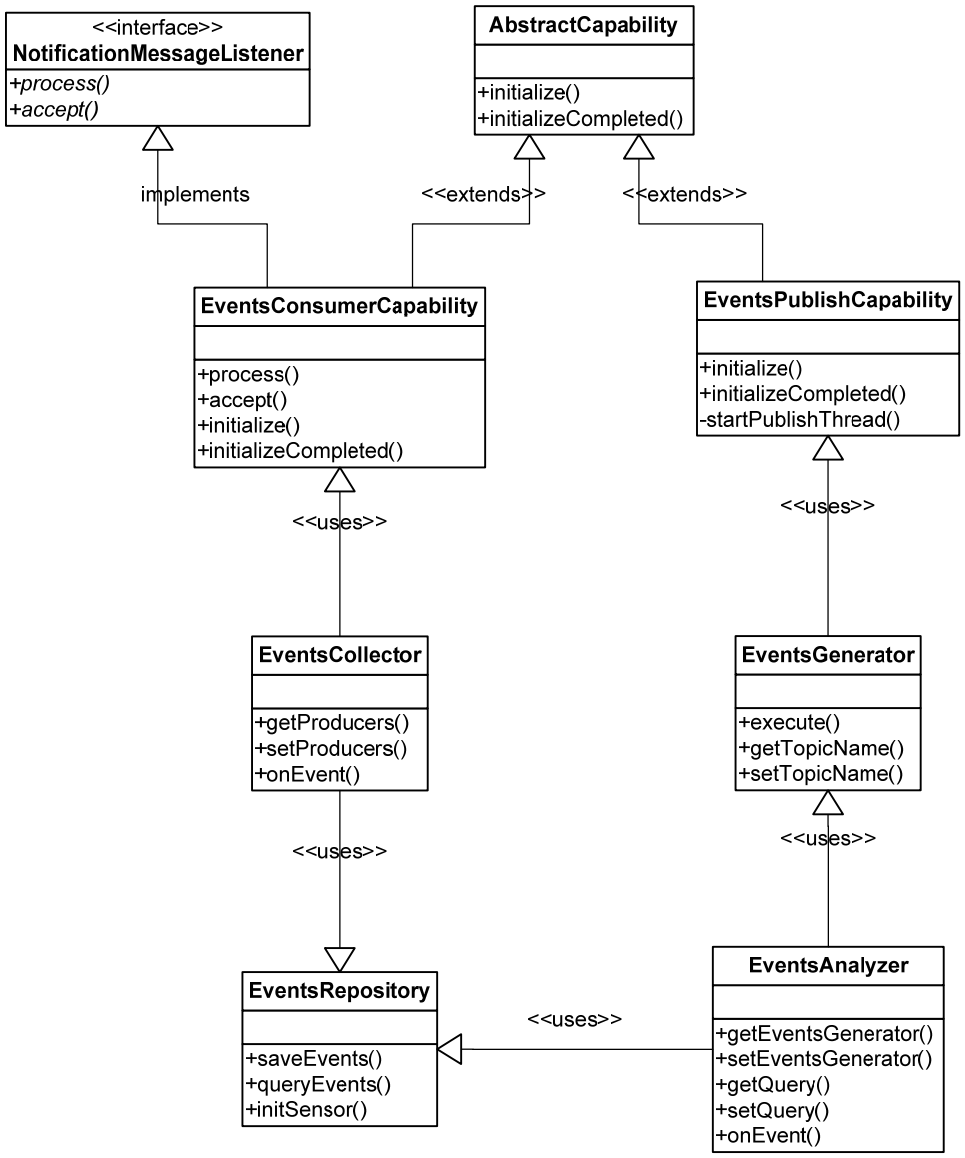

Figure 4-3 Sensor Components Class Diagram

4.2 Implementation environment and tools

The infrastructure of the runtime environment is shown in Figure 4-4.

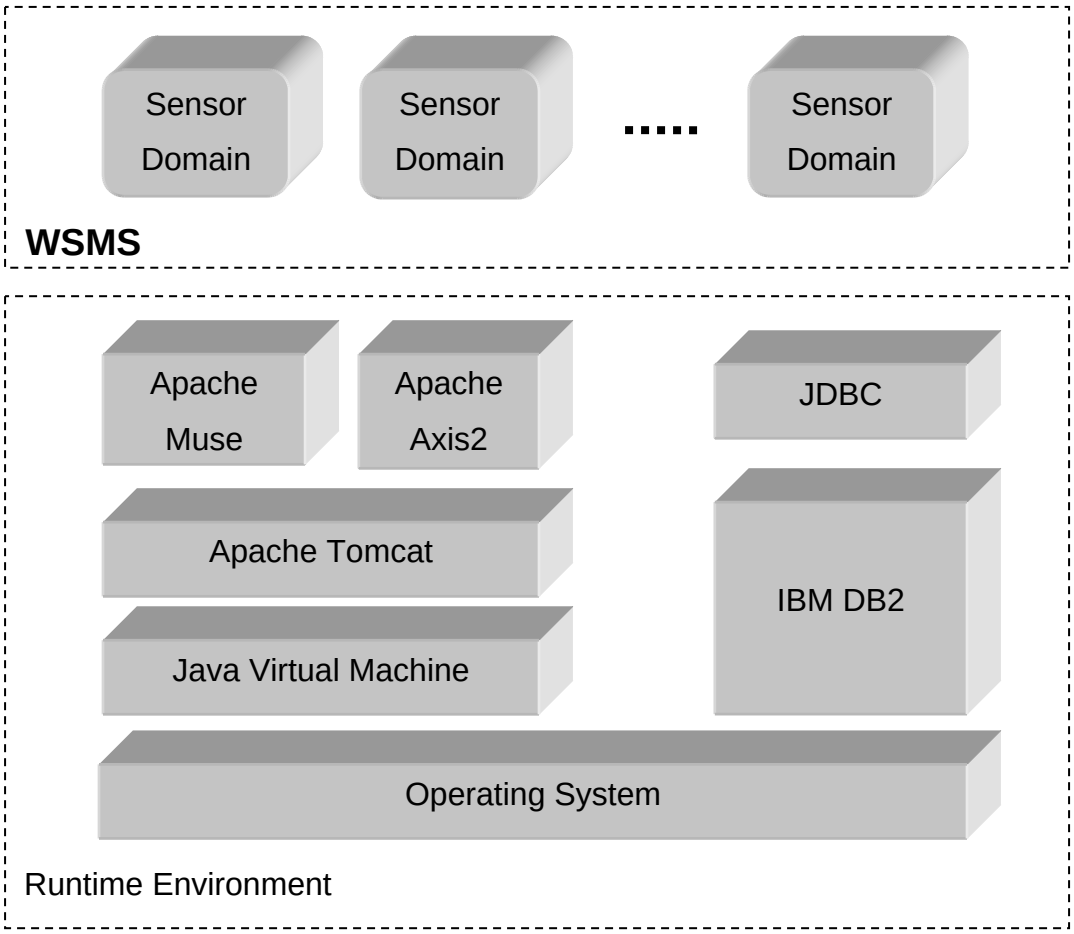

Figure 4-4 WSMS Runtime Environment

The WSMS prototype is built on the Microsoft Windows XP platform. The application is developed using the Java Enterprise Edition (JEE) [24] and runs on Apache Tomcat [27]. Eclipse Test and Performance Tools Platform (TPTP) [29] is an open platform upon which developers can build unique monitoring tools that easily integrate with Eclipse and other tools. It contains

WSDM tooling to help generate WSDL and other configuration files. Apache Muse [26] and Axis2 [25] are employed to set up a WSDM environment. IBM DB2 is used to save the events in XML format. JDBC is used to support the interface between application and database. This section provides an introduction to the implementation tools used in the prototype of WSMS.

4.2.1 Eclipse TPTP and Build to Manage

Eclipse [28] is an integrated development environment that is mainly focused on building Java, Web Service and Web applications. Eclipse TPTP [29] is a powerful framework integrating with monitoring tools such as Build to Manage (BtM) [29] tool kits for WSDM. BtM toolkits make it easy for developers to add manageability to distributed Java applications. BtM primarily provides tooling for the modeling of resource manageability characteristics, generation of WSDM compliant endpoints, and deployment/testing of developed endpoints. We use BtM to generate WSDM endpoints, add the implementation code and deploy the built endpoints.

4.2.2 Apache Muse

Apache Muse is a Java-based implementation of the WS Resource Framework (WSRF), WS BaseNotification, and WS Distributed Management specifications. It is a framework upon which users can build Web Service interfaces for manageable resources without having to implement all of the "plumbing" described by the aforementioned standards. Applications built with Muse can be deployed in Apache Axis2 environments, and the project includes a set of command line tools that can generate the proper artifacts for a deployment scenario. Apache Muse is already integrated in Eclipse TPTP. It works as a plug-in with a Graphic User Interface (GUI) editor to model Web Service projects.

4.2.3 Apache Tomcat

Apache Tomcat is the JEE Servlet container that is used in the official Reference Implementation for the Java Servlet and Java Server Pages technologies [24]. It runs on Java Virtual Machine (JVM) and supports Apache Muse as well as Axis2. WSMS is supported by Muse and Axis2 and runs on Apache Tomcat. Multiple Sensor domains can run at the same time to manage and monitor various resources. They can run on the same machine, or physically distributed machines as long as they can connect via Web Service references.

4.2.4 IBM DB2

IBM‘s DB2 is employed as the events repository because It supports pure XML as well as XQuery. With a DB2 JDBC library, WSMS is able to access the events repository.

4.3 Scenario

Here we give a brief reminder on the scenario we introduced in Chapter 3. We have two components being monitored: the DBMS and the Web Service. Each has multiple properties to indicate the performance, such as buffer pool hit rate, throughput, rejection rate and workload mix. Good performance is characterized by a high hit rate, high throughput and a low reject rate. When the WS workload intensifies, or there is a shift in the workload mix, the system resources may be taxed leading to performance degradation. The hit rate and throughput may decline or the rejection rate may increase. To correct the situation, the management entity must be made aware of the conditions and make an informed decision regarding adjustments to the amounts of resources, which are buffer pool size and connection pool for the DBMS and Web Service, respectively.

In this scenario, WSMS resides between the monitored components and the management entity. The Policy Manager is responsible for distributing this policy to the Sensor Manager. The Sensor

Manager installs a group of Sensors to monitor the Database and the Web Service components by collecting metrics, generating events and enforcing the Sensor policies (which generate complex events when matches are found). At the top level of the hierarchy, WSMS sends complex events to the management entity to suggest that new resources may be needed.

4.3.1 Designing a hierarchy of Sensors

A hierarchy of Sensors shown in Figure 3-5 collects correlates and processes the messages from low level Sensors that monitor the actual components to the highest level of Sensor that processes complex events arriving from several sources.

The lowest level of the Sensor hierarchy includes 4 Sensors; DBHitRateSensor, WSCallSensor, WSThroughputSensor and WSRejectRateSensor. They subscribe to the topics illustrated in Figure 4-5 to receive messages from the endpoints, AWSEdb and AWSEws. The higher level Sensors in AWSE are DBSensor and WSSensor. The top level Sensors process events sent from lower level of the four Sensors and publish complex events to the management entity.

4.3.2 Designing Domain and Topics

Given these six Sensors, we decide to put them into one domain to simplify the prototype. Therefore, we only need to create one Sensor Manager to manage six Sensors. Additionally, we need to create a Policy Manger to distribute policy.

As we discussed in section 4.1, these two are WSDM endpoints. Policy Manager has its URI: http://localhost:8080/PolicyManager/services/PolicyManager. Sensor Manager has http://localhost:8080/SensorManager/services/SensorManager as its URI. Policy Manager defines the topic POLICY_NOTIFICATION through which to notify the Sensor Manager to retrieve new policies. The details of the policies are discussed in the following section.

Moreover, we need to identify the topics and producer URIs used in the domain for Sensors to consume events. In Figure 4-5, we have two external event producers, AWSEdb and AWSEws, which have the URIs: http://localhost:8080/AWSEdb/services/AWSEdb and http://localhost:8080/AWSEws/services/AWSEws, respectively. All of the Sensors share the same producer URI with Sensor Manager defined above. Their topics are shown in Table 4-2.

Table 4-2 Topics Used in the Scenario

Producer	Consumer	Topic
AWSEdb	DBHitRateSensor	HITRATE
AWSEws	WSCallSensor	WSCALLMIX
AWSEws	WSThroughputSensor	THROUGHPUT
AWSEws	WSRejectRateSensor	REJRATE
DBHitRateSensor	DBSensor	DBHitRateTopic
WSCallSensor	DBSensor	WSCallTopic
WSThroughputSensor	WSSensor	WSThroughputTopic
WSRejectRateSensor	WSSensor	WSRejectRateTopic
DBSensor	Management Entity	DBTopic
WSSensor	Management Entity	WSTopic

After confirming these properties, we use Eclipse TPTP and WSDM Tooling to automatically create the Sensor Manager and the Policy Manager. Appendix D lists the Muse configuration files used for our scenario. It defines the context of the endpoints as well as the capabilities of them. Now that we have Sensor Manager and Policy Manager implemented, we need to design the policy list for the Sensor Manager in order to install the Sensors.

4.3.3 Designing Policies

In our scenario, we have six Sensors in the domain. Accordingly, there are six policies that define the Sensors in the domain. Following the schema in Section 3.5, we first map the components of each Sensor to the parts of policy. The location of the input events is defined for the Sensor's Event Collector. The Event Pattern is specified for the Events Analyzer and the Events Generator uses the Event Output. The attributes of a policy are defined as follows:

- Event Source includes the events source URI and topic names.
- Pattern Statement is a statement written in EPL, namely XQuery.
- Event Body is the event contents to be published in the event the Sensor detects a pattern match.
- Event Destination is topic names to which the events are published.

The policies for each of the Sensors are described as follows. Detailed policy is shown in Appendix C.

DBHitRateSensor

- Subscribes to the topic of "HITRATE", which is published by the monitored system "AWSEdb".
- Detects when the hit rate decreases more than 10 percent in 30 seconds. The XQuery statement is represented as:

```
xquery
declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";
declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ;
for $a in db2-fn:xmlcolumn('DBHITRATESENSOR.EVENT')/muws1:ManagementEvent,
    $b in db2-fn:xmlcolumn('DBHITRATESENSOR.EVENT')/muws1:ManagementEvent
where
```

xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and xs:dateTime(fn:string($b/@ReportTime)) < xs:dateTime(fn:string($a/@ReportTime)) + xdt:dayTimeDuration("PT30S")

and fn:number($a/muws2:Situation/muws2:Message) - fn:number($b/muws2:Situation/muws2:Message) > 10

return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>;

Once a match is found, this statement returns the event IDs which caused the match.

- Sends an event including a plain text of "Hit rate is declining" in the message body.
- Publishes the above event to the topic "DBHitRateTopic"

WSCallSensor

- Subscribes to the topic "DBMSCall", which is published by the monitored system "AWSEws".
- Detects when the number of Web Service calls increases more than 5 percent in 60 seconds. The XQuery statement is represented as:

xquery

declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";

declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ;

for $a in db2-fn:xmlcolumn('WSCALLSENSOR.EVENT')/muws1:ManagementEvent

, $b in db2-fn:xmlcolumn('WSCALLSENSOR.EVENT')/muws1:ManagementEvent

where

xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and xs:dateTime(fn:string($b/@ReportTime)) < xs:dateTime(fn:string($a/@ReportTime)) + xdt:dayTimeDuration("PT60S")

and fn:number($b/muws2:Situation/muws2:Message) -fn:number($a/muws2:Situation/muws2:Message) > 5

return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>

Once a match is found, this statement returns the event IDs which cause the match.

- Sends an event saying “DBMS Call is increasing”
- Publishes the above event to the topic “DBHitRateTopic”

WSThroughputSensor

- Subscribes to the topic “THROUGHPUT”, which is published by the monitored system “AWSEws”.
- Detects the value of throughput increasing more than 10 transactions per second in 30 seconds. The XQuery statement is represented as:

  xquery

  declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";

  declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ;

  for $a in db2-fn:xmlcolumn('WSTHROUGHPUTSENSOR.EVENT')/muws1:ManagementEvent,

  $b in db2-fn:xmlcolumn('WSTHROUGHPUTSENSOR.EVENT')/muws1:ManagementEvent

  where

  xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime))

  and xs:dateTime(fn:string($b/@ReportTime)) < xs:dateTime(fn:string($a/@ReportTime)) + xdt:dayTimeDuration("PT30S") and fn:number($b/muws2:Situation/muws2:Message)-fn:number($a/muws2:Situation/muws2:Message)>10

  return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>

  Once a match is found, this statement returns the event IDs which cause the match.
- Sends an event saying “Throughput increases”
- Publishes the above event to the topic “WSThroughputTopic”

WSRejectRateSensor

- Subscribes to the topic “REJRATE”, which is published by the monitored system “AWSEws”.

- Detects that the reject rate has increased in the last 20 seconds. The XQuery statement is represented as:

  xquery

  declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";

  declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ;

  for $a in db2-fn:xmlcolumn('WSREJECTRATESENSOR.EVENT')/muws1:ManagementEvent,

  $b in db2-fn:xmlcolumn('WSREJECTRATESENSOR.EVENT')/muws1:ManagementEvent where

  xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and

  xs:dateTime(fn:string($b/@ReportTime)) < xs:dateTime(fn:string($a/@ReportTime)) +

  xdt:dayTimeDuration("PT20S") and fn:number($b/muws2:Situation/muws2:Message)-

  fn:number($a/muws2:Situation/muws2:Message)>0

  return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>

  Once a match is found, this statement returns the event IDs which cause the match.

- Sends an event saying “Reject rate increases”
- Publishes the above event to topic “WSRejectRateTopic”

DBSensor

- Subscribes to two topics,“DBHitRateTopic” and “WSCallTopic”, which are published by the two Sensors, DBHitRateSensor and WSCallSensor, respectively.
- Detects if there are events from both DBHitRateSensor and WSCallSensor. This situation means that the hit rate of DBMS is declining and the percentage of OLAP calls is increasing at the same time. The XQuery statement is represented as:

  xquery

  declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";

  declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ;

  for $a in db2-fn:xmlcolumn('DBSENSOR.EVENT')/muws1:ManagementEvent

  , $b in db2-fn:xmlcolumn('DBSENSOR.EVENT')/muws1:ManagementEvent

where $a/muws2:Situation/muws2:Message/text()='Hit rate is declining'

and $b/muws2:Situation/muws2:Message/text()=DBMS Call is increasing'

return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>

Once a match is found, this statement returns the event IDs which cause the match.

- Sends an event saying “BP Size + 1000”, which suggests the buffer pool size needs a new 1000 kb to be added to increase the performance.
- Publishes the above event to topic “DBTopic”

WSSensor

- Subscribes to two topics, “WSThroughputTopic” and “WSRejectRateTopic”, to which two Sensors, WSThroughputSensor and, WSRejectRateSensor respectively publish events.
- Detects if there are events from both WSRejectRateSensor and WSThroughputSensor. This situation means that the reject rate is increasing and throughput is increasing at the same time indicating an increased load on the system. The XQuery statement is represented as:

  xquery

  declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";

  declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ;

  for $a in db2-fn:xmlcolumn('WSSENSOR.EVENT')/muws1:ManagementEvent

  , $b in db2-fn:xmlcolumn('WSSENSOR.EVENT')/muws1:ManagementEvent

  where $a/muws2:Situation/muws2:Message/text()='Throughput increases'

  and $b/muws2:Situation/muws2:Message/text()='Reject increases'

  return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>

  Once a match is found, this statement returns the event IDs which cause the match.

- Sends an event saying “Pool Size + 5”, which suggests the connection pool needs another 5 new connections to be added into the pool.
- Publishes the above event to topic “WSTopic”

4.3.4 Prototype Testing

In this section we step through the prototype, from normal operations through changes that drive the complex events. We can see how WSMS behave as the load on the resources changes. Then we discuss some cases in the scenario to evaluate WSMS.

4.3.5 Running through prototype

In our scenario, we can manually adjust the OLAP/OLTP call mix to affect other three performance metrics, DB hit rate, WS throughput and WS reject rate, indirectly. For example, if we increase OLAP calls, the hit rate will decrease while the throughput and reject rate will both increase because OLAP calls consume much more resources than OLTP. With all of this information, we can tell that the performance degrades. Under this situation, we need to allocate more resources such as DB buffer pool size to improve the performance. WSMS can be employed here to monitor the systems and detect this situation automatically.

When the two systems are launched, everything runs smoothly until time t. At time t, we increase the percentage of OLAP calls t from 0 to 25%. Then we increase the percentage of OLAP calls to 50% at time t+10s and then keep it constant for the rest of the experiment. This trend is shown in Figure 4-6.

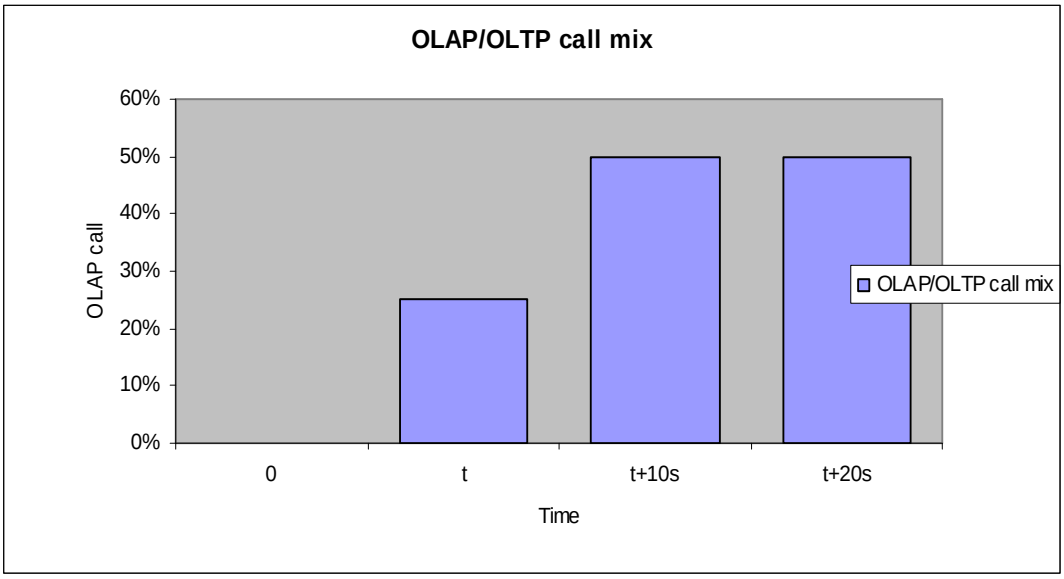

Figure 4-5 OLAP/OLTP Call Mix

The two components become slow because more resources are needed to handle the increasing OLAP calls. Correspondingly, the other three metrics change.

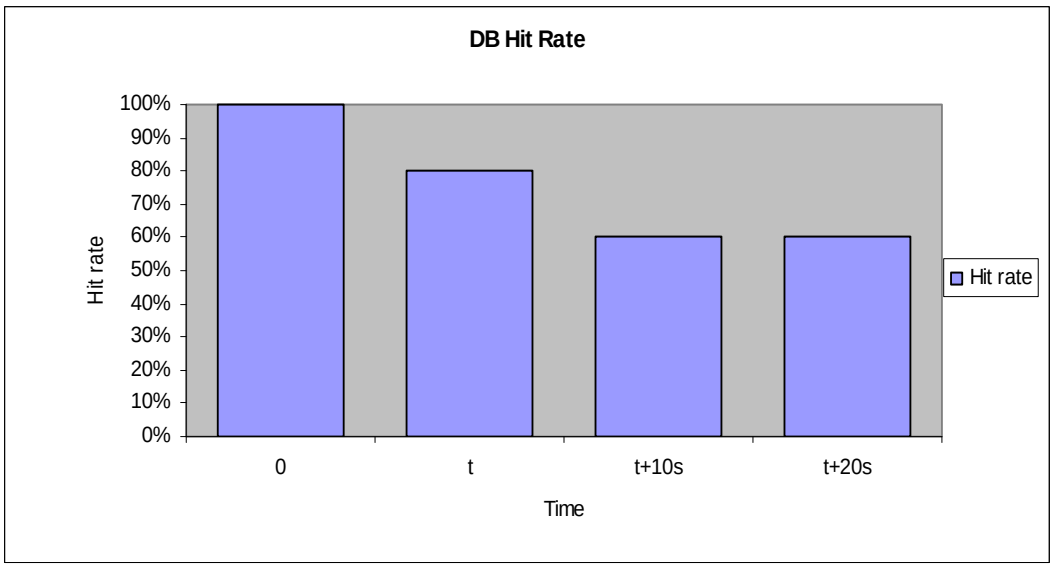

Figure 4-6 DB Hit Rate

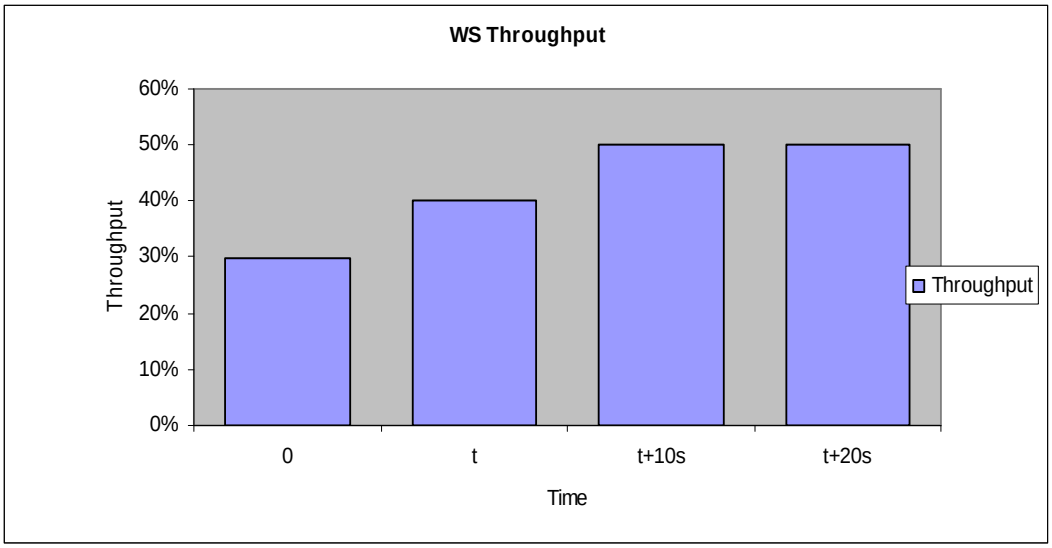

Figure 4-7 WS Throughput

Figure 4-8 WS Reject Rate

WSCallSensor detects the change of OLAP calls and fires off an event representing this situation to DBSensor. DBHitrateSensor detects the decline of DB hit rate and sends an event to DBSensor about it as well. DBSensor therefore detects that the DB workload has changed and sends an

event to the management entity asking for more resources. On the other side, WSRejectRateSensor detects that the rise in the reject rate and publishes an event about this change to WSSensor. WSThroughputSensor also finds out that the throughput increases, which indicates an increase in the arrival rate. It sends an event to WSSensor too. Similar to DBSensor, when WSSensor receives the two events, it sends an event to the management entity suggesting that more resources are needed to improve the performance.

The management entity's responsibility is to take actions to react to the problems Sensors detect, which is to allocate more resource in our case. This is out of our scope in the thesis.

From the prototype, WSMS monitors DB and WS systems running smoothly. Once some problems happen, WSMS is able to detect the situation automatically and inform the manager of the problems.

4.3.6 Event Processing Cases

WSMS's capabilities to process events depend on the information carried within the event message. In our work we use the following two elements of a WSDM event:

1. Timestamp. This is a formatted field to represent the exact time when an event occurred.
2. Message body. This is a text string that describes the details of the event, such as "hit rate is 90" or "The percentage of OLAP Web Service calls is declining".

In order to simplify our scenario, we name the timestamp in an event as event.time and the message body in an event as event.value. In our scenario, we find WSMS capable of doing the following three cases.

Detect changes over a time window

AWSEdb sends messages to the DBHitRateSensor regularly with the DBMS hit rate. The hit rate changes constantly due to changes in the database workload. The Sensor collects and saves the hit rate data events as tuples in the repository. The Sensor uses its policy to decide when to generate a complex event. For instance, the event pattern for DBSensor is "Detect if the Hit rate declines more than 10% in 60 seconds". The Sensor saves all events into repository and executes an XQuery each time a new event arrives. As shown in Figure 4-10, the XQuery virtually holds a time window of 60 seconds to scan the events repository to compare the hit rate value one by one.

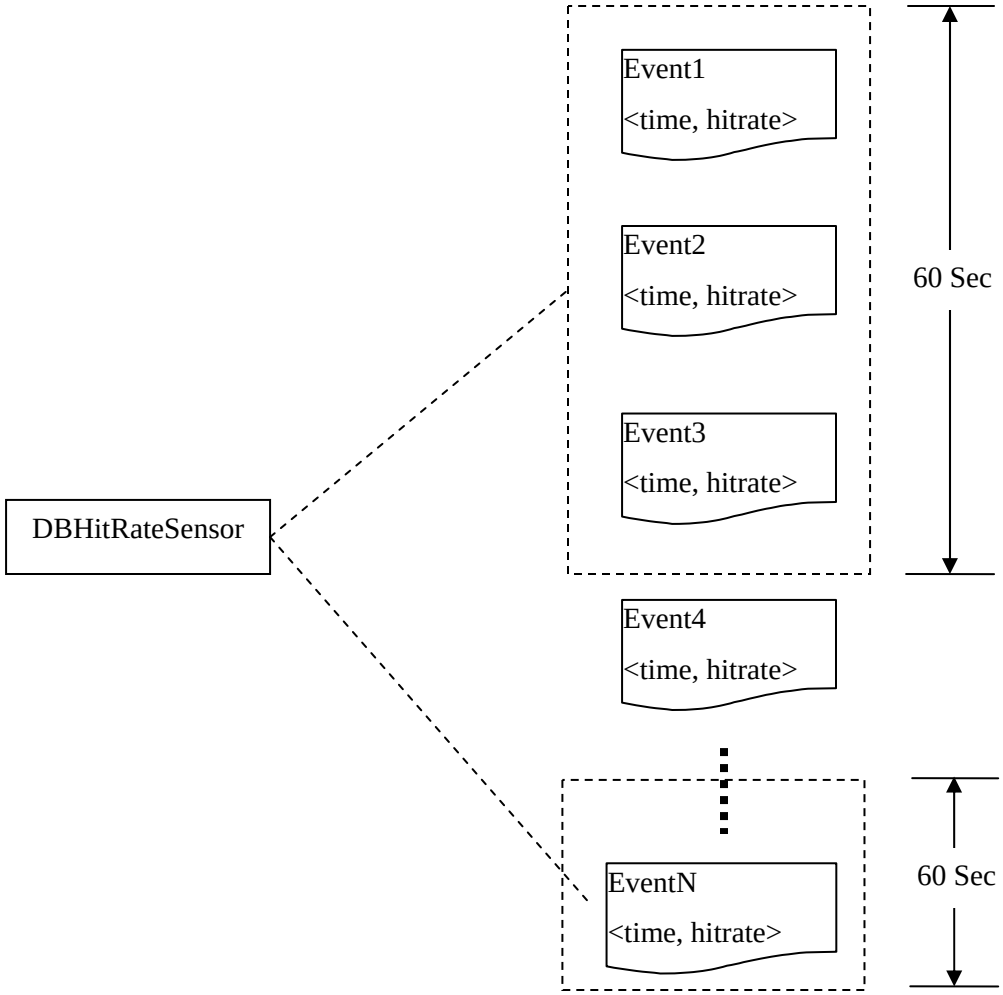

Figure 4-9 Detecting Changes in a Time Window

According to the event patterns, the Sensor has the following pseudo code:

Select event_N-1, event_N

from repository

where event_N.time – event_N-1.time < 60 seconds

and event_N.time > event_N-1.time

and event_N.hitrate – event_N-1.hitrate > 10

The Sensor submits the above XQuery to the database. Once it finds a match, it returns a set of event IDs representing the events that cause the match and generates and fires a complex event.

Detect changes over a sequence of events

AWSEws sends messages to WSCallSensor regularly with the metrics of Web Service call. This value changes with changes in the workload. The Sensor collects and saves the values in the repository and evaluates using its policy to find matches. In this case, the policy is "Detect if the percentage of OLAP calls increases by more than 5% over 4 events". As shown in Figure 4-11, the Sensor iterates the events with a sliding window, grouping them in fours and compares the value differences.

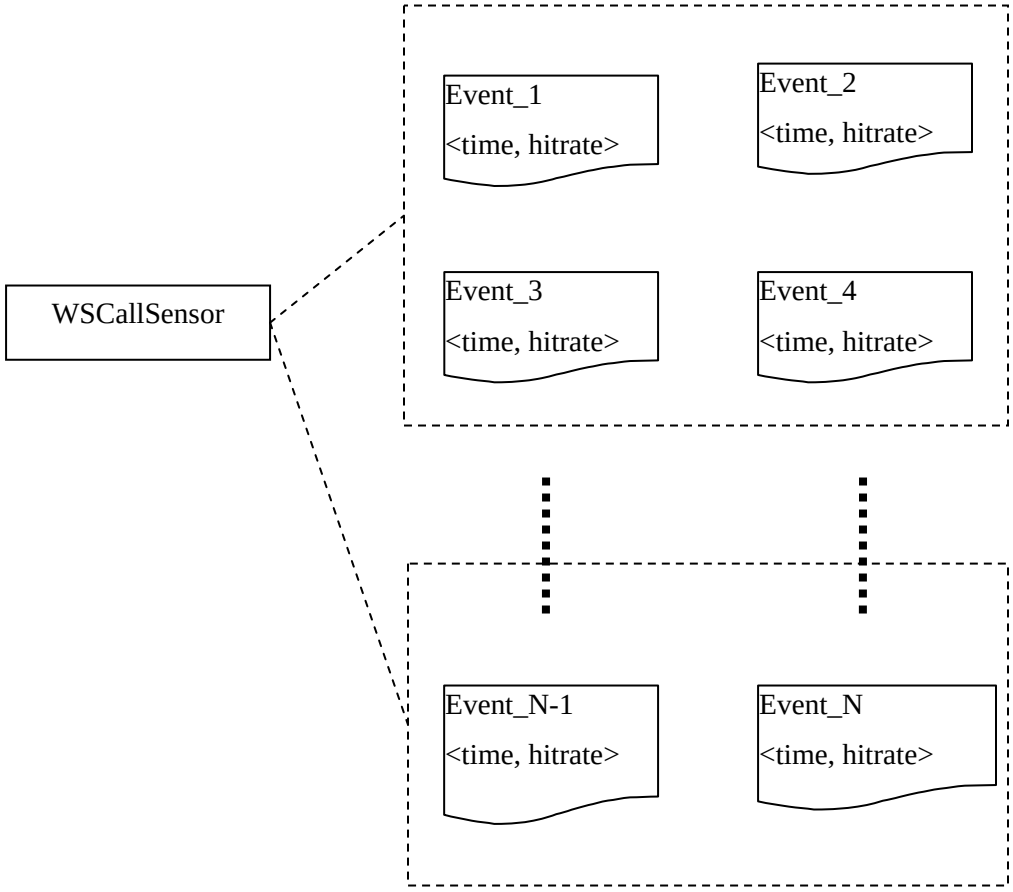

Figure 4-10 Detect Changes Over a Sequence of Events

According to the event pattern, the Sensor has the following pseudo code:

*Select **

from repository

where event_N-2.time > event_N-3.time

and event_N-1.time > event_N-2.time

and event_N.time > event_N-1.time

and (event_N-2.hitrate–event_N-3.hitrate>5

or event_N-1.hitrate –event_N-2.hitrate >5

or event_N.hitrate –event_N-1.hitrate>5)

The Sensor submits the above Xquery to the database. Once it finds a match, it returns a set of event IDs, which represent the events that caused the match. A complex event is generated and published.

Collect events from multiple sources and detect if an event exists

As a higher level Sensor, DBSensor consumes events from multiple sources, in our case from DBHitRateSensor and WSCallSensor. The rule is "Detect if one event from DBHitRateSensor and one event from WSCallSensor exist". We use a content-based search to check where an event comes from. For example, if an event is from the DBHitRateSensor, the value of the event should be exactly equal to "Hit rate is declining". This message is specified by DBHitRateSensor. Therefore, the Sensor has the following pseudo code:

Select event_M, event_N

from repository

where event_N.value='Hit rate is declining'

and event_M.value=' DBMS Call is increasing'

Once it finds a match, it returns a set of event IDs, which represent the events that caused the match. A complex event is generated and published. In this case, the event sources physically reside on the same server, or site. The information within the WEF allows us to distinguish events from multiple sites. In WEF, there is a tag named "SourceComponent" which includes the source component's locations or other details. The XQuery may be adapted to resemble the following:

Select event_M, event_N

from repository

where event_M.value='Hit rate is declining'

and event_N.value=' DBMS Call is increasing'

and event_M.source = event_N.source.

4.4 Summary

In this chapter we present the implementation of WSMS. A discussion of the implementation environment and tools used was then provided followed by the details of WSMS implementation, followed by a prototype testing and evaluation for WSMS. This evaluation defines the scope of WSMS's capability in terms of events processing. Essentially, it largely depends on the events format. In our scenario, we are using the WSDM Event Format, which correlates events by timestamp. Therefore it is able to detect a sequence of events in a time window. Additionally, it can distinguish events from multiple sources, or sites. This is significant because the environment is distributed.

Chapter 5
Conclusions

5.1 Summary

Today's information systems are becoming increasingly complex and difficult to manage. Web Service standards and development platform provides a method to loosely decouple systems and simplify interoperability among systems and components. The WSDM standards provide a standard interface for the monitoring and management of Web Service enabled systems which employ events to represent the situation data. WSMS utilizes these events to monitor components using a Sensor-based approach. Using Complex Events Processing, Sensors in WSMS detect patterns in a sequence of events or in a time window. Therefore, WSMS is able to monitor WSDM endpoints and provide suggestions for decision-making systems by processing events.

One remarkable importance of our approach is that we employ many standards in the implementation. For instance, we use XQuery as the EPL to describe event patterns. XQuery is recommended by W3C who develops Web standards. Another example is WEF, standardized by OASIS who regulates the most Web Service standards. Conforming standards improves accessibility, Interoperability and maintainability of our approach. Additionally, WSMS is extensible and flexible. Individual Sensors are small units so that they can be distributed in diversified situations.

This thesis provides an exploration of WSMS incorporating CEP and policy based design as the two key technologies. A Sensor in WSMS implements a CEP model, aggregating events, running rules against them and taking action by producing a complex event. Hierarchies of Sensors are

formed to aggregate events from multiple sources. A Sensor's behavior is abstracted to a pattern which is expressed as a policy. Policy Managers and Sensor Managers work together to distribute policy. Sensor Manager is responsible for managing Sensor's life cycle.

Prototyped in this thesis was an instance of WSMS monitoring a Web Service environment including DBMS and Web Service components. The monitored systems are represented by WSDM endpoints that produce events indicating their current performance or status. Six Sensors are placed in the same WSDM environment to detect abnormal situations. Each Sensor includes an event pattern written in XQuery that is executed whenever a new event comes in. If the defined sequence of events is detected, a complex event is sent to a higher level Sensor. The top level Sensor reports directly to an Analyzer of a decision-making system.

5.2 System limitations and Future work

The main objective of WSMS presented in this thesis was a proof of concept of the Sensor-based approach to system monitoring. Within the design process, a number of issues were overlooked and assumptions were made. The following presents these issues and assumptions as areas of future work.

- The implementation of WSMS does not support multiple domains. In our prototype, there is only one domain which manages six Sensors. A more complicated situation is two domains including the Sensors separately. Every domain has a Sensor Manger maintaining a policy list corresponding to the Sensors within it. The way for every Sensor Manager to get its own policy list is one of the future goals. Otherwise, once Sensors are set up and initialized by Sensor Manager, they run exactly like existing only one domain.

- The event patterns are limited to be based on time. The WSDM Event Format defines a timestamp in the message body, which is utilized by WSMS as a correlation element. XQuery is able to manipulate timestamps in XML such as detecting patterns within in a time window. However, the other two correlation types, causality and aggregation cannot be applied here because WSDM Event Format is only correlated by time. A future approach may be to add extension elements to the WSDM Event Format to implement these two correlation types. XQuery can be employed as the EPL because it is capable of handling all of the three correlation types.
- WSMS does not include a GUI editor for policy and hierarchy creation. All of the Sensor entities in WSMS are based on policy, which is managed by the Sensor Manager and distributed by the Policy Manager. The prototype in this thesis assumes that the policy is ready to be deployed at run time. A GUI editor for policy as well as hierarchy creation is expected to make policy changes dynamic. A tool to allow users to monitor the incoming/outgoing messages for each Sensor may also be useful.

References

1. Boag, S., Chamberlin, D., F. Fernández, M.F., Florescu, D., Robie, J. and Siméon, J., *XQuery 1.0: An XML Query Language*. August 10 2008, http://www.w3.org/TR/xquery/.
2. Box, D, Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., Kaler, C., Langworthy, D., Leymann, F.,Lovering, B., Lucco, S., Millet, S., Mukhi, N., Nottingham, M., Orchard, D., Shewchuk, J., Sindambiwe, E., Storey, T., Weerawarana, S. and Winkler, S., *Web Service Addressing*, June 28 2008, http://www.w3.org/Submission/ws-addressing/.
3. Bryan, D., Draluk, V., Ehnebuske, D., Glover T., Hately, A.,Husband, Y.L., Karp, A., Kibakura, K, Kurt, C., Lancelle, J., Lee, S., MacRoiird, S., Manes, A.T., McKee, B., Munter, J., Nordan, T., Reeves, C, Rogers, D., Tomlinson, C., To, C., Riegen, C.V. and Yendluri, P., *Universal Description Discovery and Integration (UDDI) Version 2*. July 7 2008, http://www.oasis-open.org/specs/index.php#uddiv2.
4. Case, J., Fedor, M., Schoffstall, M. and Davin, C., *A Simple Network Management Protocol (SNMP)*. RFC 1098. April, 1989.
5. Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S., *Web Service Description Language (WSDL) 1.1*, July 20 2008, http://www.w3.org/TR/wsdl.
6. Graham, S., Hull, D. and Murray, B, *WS Notification Version 1.3*, July 2 2008, http://www.oasis-open.org/specs/index.php#wsnv1.3.
7. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A. and Lafon, Y., *Simple Object Access Protocol (SOAP) 1.2,* August 2 2008, http://www.w3.org/TR/soap/.

8. IBM, *An architectural blueprint for autonomic computing.* August 1 2008, http://www.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf
9. IBM. *Autonomic Computing*, 2008; August 15 2008, http://www.ibm.com/developerworks/autonomic.
10. IBM, *DB2 9*, August 10 2008, http://www-306.ibm.com/software/data/db2/9/.
11. Keller, A. and Ludwig, H., *the WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Service.* Journal of Network and Systems Management, 2003, volume 11.
12. Kreger, H., Bullard, V. and Vambenepe, W. *WSDM Management Using Web Service Version 1.0 (WSDM-MUWS)*, August 2 2008, http://www.oasis-open.org/specs/index.php#wsdm-muwsv1.0.
13. Kreger, H., Wilson, K. and Sedukhin, I., *WSDM Management of Web Service Version 1.0 (WSDM-MOWS)*, August 5 2008, http://www.oasis-open.org/specs/index.php#wsdm-mowsv1.0.
14. Lawrence, K., Kaler, C., Nadalin, A., Monzillo, R. and Hallam-Baker, P., *Web Service Security: SOAP Message Security 1.1*, July 2 2008, http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.
15. Luckham, D., *the Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. 2002, Addison-Wesley.
16. Maullo, M.J. and Calo, S.B., *Policy Management: An Architecture and Approach*, in Proceedings of the IEEE First International Workshop on Systems Management, Los Angeles, California, USA, April 14-16 1993.

17. Microsoft, *Microsoft Message Queuing*, August 1 2008, http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx.
18. OASIS, *Web Service Distributed Management Version 1.1*, August 2 2008, http://www.oasis-open.org/specs/index.php#wsdmv1.1.
19. Papazoglou, M.P., *Service-oriented computing: concepts, characteristics and directions*, in Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003.
20. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D. and Traverso, P., *Planning and Monitoring Web Service Composition.* In ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services, Whistler, British Columbia, Canada, June 3-7, 2004.
21. Renesse, R.V., Birman, K. and Vogels, W., *Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining*, ACM Transactions on Computer Systems, Vol.21, No. 2, pp 164-206, May 2003.
22. Sahai, A., Machiraju, V., Sayal, M., Li, J.J. and Casati, F, *Automated SLA Monitoring for Web Service.* In Proc. of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM, Montreal, Canada, October 2002.
23. Stephen, E.H. and Atkins, E.T., *Automated System Monitoring and Notification with Swatch*, in Proceedings of the 7th USENIX conference on System administration, USENIX Association: Monterey, California, USA, 1993.
24. Sun Microsystems. *Java Enterprise Edition*, August 10 2008, http://java.sun.com/javaee/.
25. The Apache Software Foundation, *Apache Axis2*, May 5 2008, http://ws.apache.org/axis2/.

26. The Apache Software Foundation, *Apache Muse*, July 12 2008, http://ws.apache.org/muse.
27. The Apache Software Foundation, *Apache Tomcat*, August 12 2008, http://tomcat.apache.org/.
28. The Eclipse Foundation. *Eclipse*. June 2 2008, http://www.eclipse.org/.
29. The Eclipse Foundation, *Eclipse TPTP*. August 2 2008, http://www.eclipse.org/tptp/.
30. Tian, W., Zulkernine, F., Zebedee, J., Powley, W. and Martin, P., *Architecture for an Autonomic Web Service Environment*, in Proceedings of the Joint Workshop on Web Service and Model-Driven Enterprise Information Systems WSMDEIS, Miami, Florida, USA, May 2005.
31. W3C, *Web Service*, July 3 2008, http://www.w3.org/2002/ws/.
32. Wikimedia Foundation, *Service Oriented Architecture*. August 1 2008, http://en.wikipedia.org/wiki/Service-oriented_architecture.
33. Zulkernine, F.H., Martin, P. and Wilson, K., *A Middleware Solution to Monitoring Composite Web Services-based Processes*, in Proceedings of 2008 IEEE Congress on Services (SERVICES'08) Part II at the Workshop on Service Intelligence and Computing (SIC) of the IEEE International Conference on Web Services (ICWS'08), pp. 149-156, IEEE CS Press, September 23-26, 2008, Beijing, China.

Appendix A
WSDM Event Schema

The WSDM Event Format is an XML format to represent management event information. The format defines a set of basic, consistent elements and attributes that allow different types of event information to be carried in a consistent manner. The WSDM Event Format provides a basis for programmatic processing, correlation, and interpretation of events from different platforms and management technologies.

The WSDM Event Format organizes event data into three basic categories, the event reporter, the event source, and extensible situation data. Each category contains a few common properties and attributes, as found in most management events. It has a flexible and extensible syntax. The following diagram shows the schema for the WSDM Event Format.

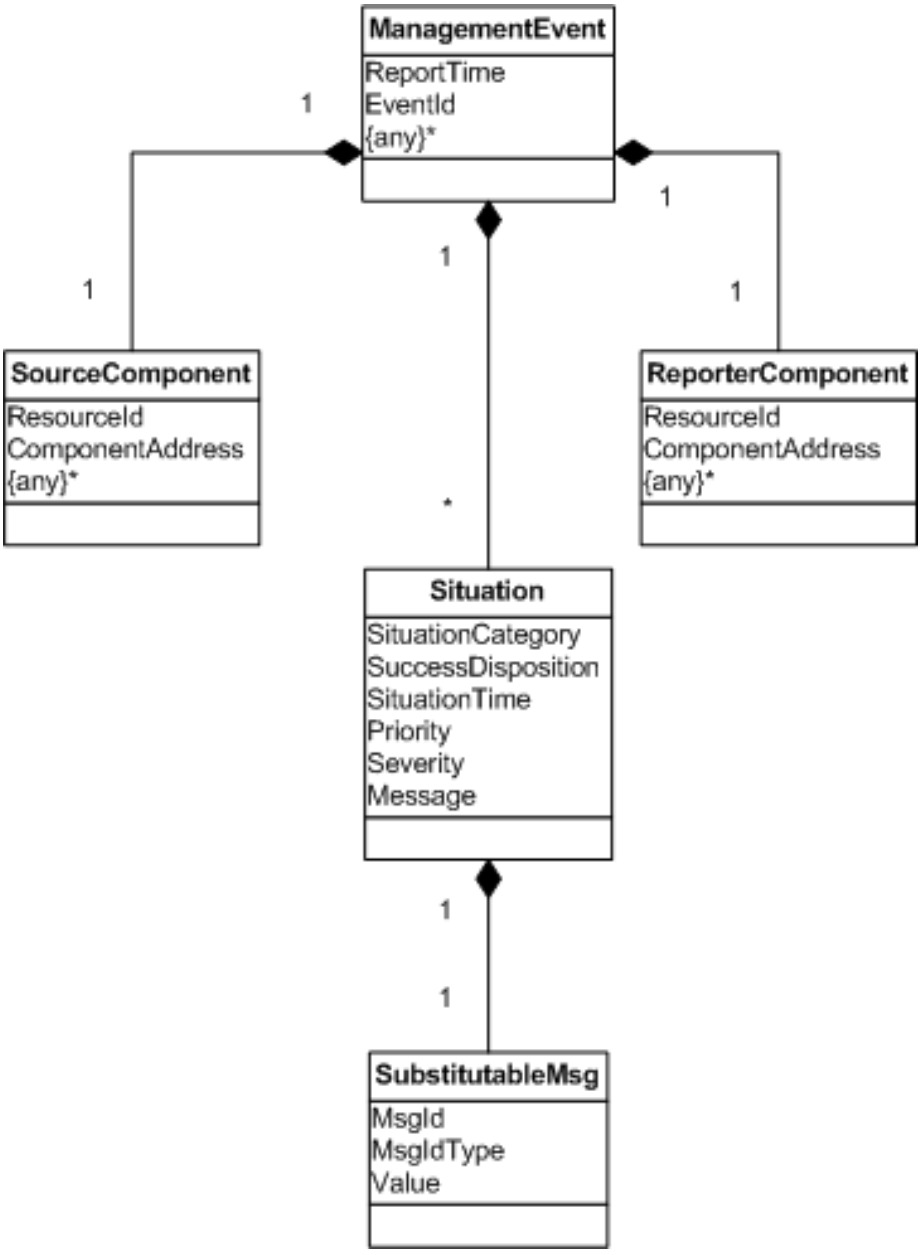

Figure A-1 WSDM Event Format Schema

Appendix B

WSMS Sequence Diagrams

Figures B-1 and B-2 show the sequence diagrams for WSMS. In B-1, there are seven roles: Policy Manager, Sensor Manager, Monitored System, Events Collector, Events Repository, Events Analyzer and Events Generator, last four of which are components of a Sensor. The Policy Manager sends a policy notification to notify the Sensor Manager, along with the URL where policy exists. The latter interprets the notification and requests the new policy as instructed by the Policy Manager. To respond to the request, the Policy Manager returns the policy to the Sensor Manager. On receiving the new policy, the Sensor Manager initializes the Sensor according to the policy: the Events Collectors are configured to subscribe to the monitored systems so that they are able to consume events sent by them; the Events Repository removes all of the history events; the Event Analyzer creates the event pattern to be queried to the repository; the Event Generator sets the topic it is going to publish events to.

Figure B-2 shows the sequence diagram at runtime after initialization. There are six roles with their lifelines in this diagram: Monitored System, the four components of one Sensor which are circled around by a dashed line and another Sensor. At runtime, the Monitored System keeps sending events to the Event Collector which saves all of them into the Events Repository. Whenever one new event is saved, the Event Analyzer submits the event pattern written in XQuery statement to the repository to find suspicious events. Once it finds, it informs the Events Generator of the problem. The latter creates an event describing the problem and sends it to another Sensor.

These two sequence diagrams describe how WSMS works to detect problems of monitored systems.

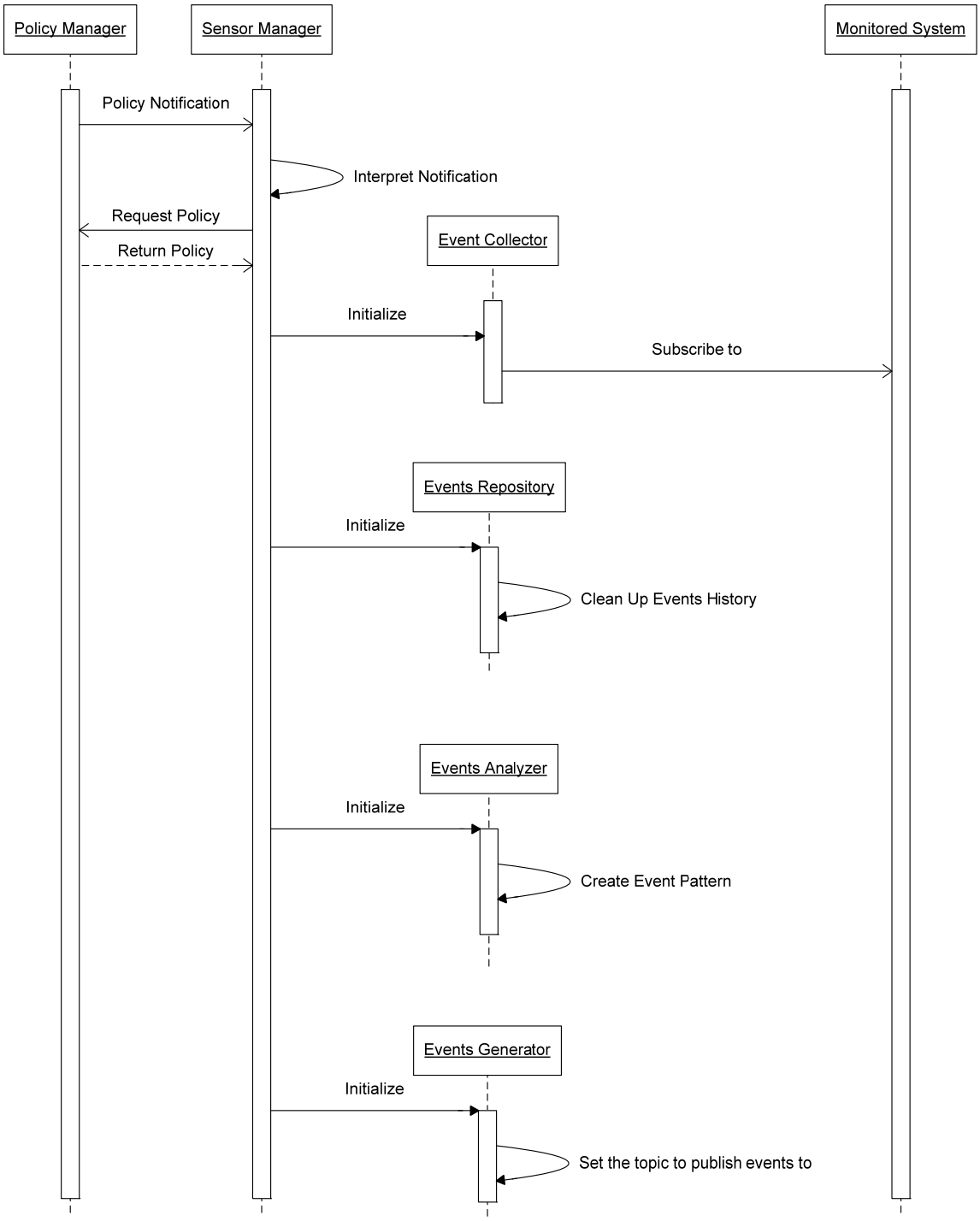

Figure B-1 Sequence Diagram at Initialization Time

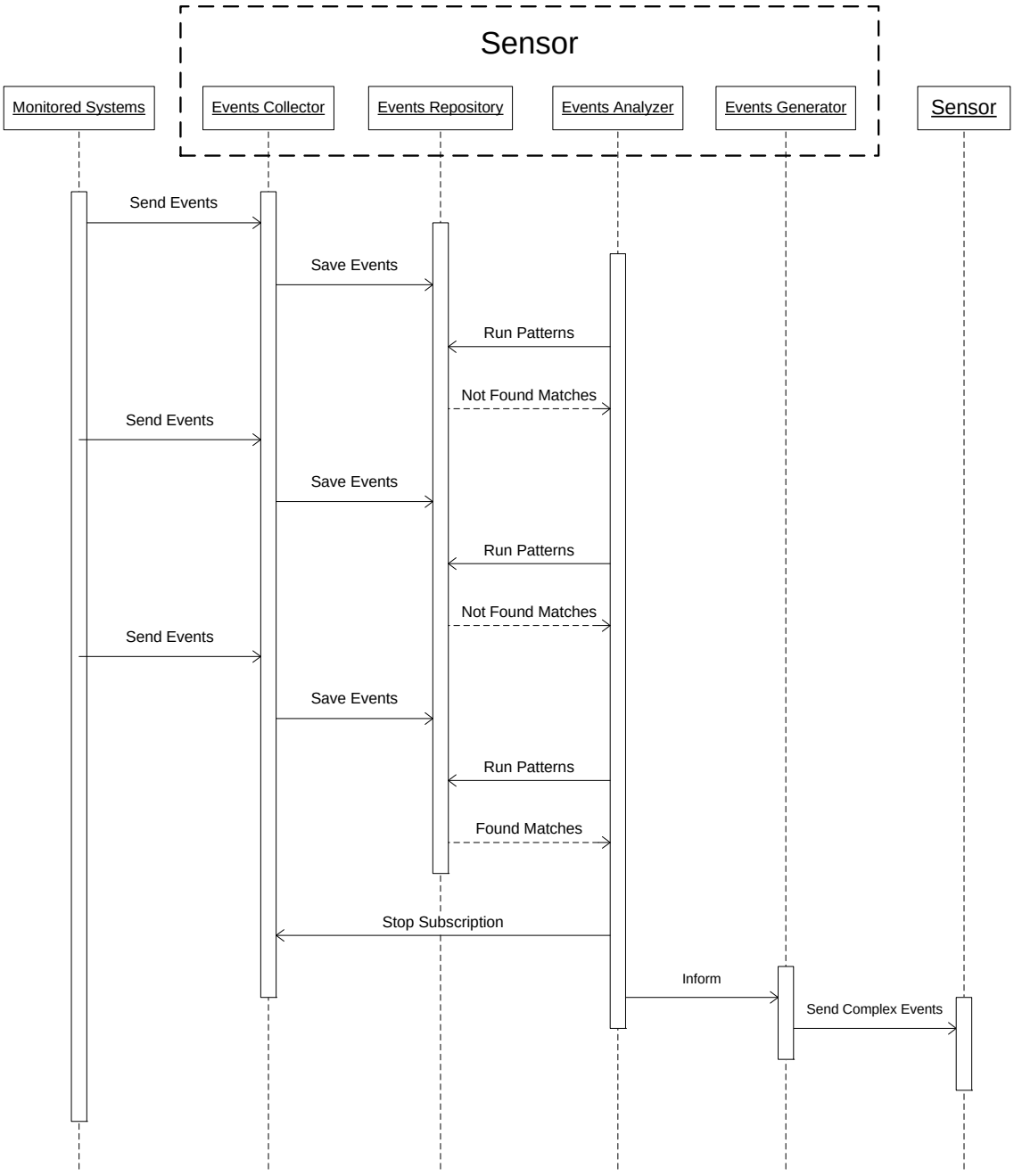

Figure B-2 Sequence Diagram at Run time

Appendix C

Policy Document

Our approach uses policy as a description of Sensor components in order to manage them in a loose-coupling manner. That is, Sensor Manager is able to retrieve policy from anywhere from external site, interpret it and control Sensor's behavior based on the policy. In our case, we use one policy file to describe six Sensors. The following is the content of the policy document used in the scenario introduced in chapter 4. .

```
<?xml version="1.0" encoding="UTF-8"?>
<ss:policy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:ss="http://awse.queensu.ca/sensor/policy/schema" >

 <ss:sensor name="DBHITRATESENSOR">
  <ss:events_collector>
   <ss:producer uri="http://localhost:8080/PolicyManager/services/PolicyManager">
    <ss:topic>
     <namespace>http://webs2.cs.queensu.ca/AWSEcap/capability/Topics</namespace>
     <name>HITRATE</name>
    </ss:topic>
   </ss:producer>
  </ss:events_collector>
  <ss:events_analyzer>
   <ss:xquery>xquery declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-
2.xsd";declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ; for $a in
db2-fn:xmlcolumn('DBHITRATESENSOR.EVENT')/muws1:ManagementEvent, $b in db2-
fn:xmlcolumn('DBHITRATESENSOR.EVENT')/muws1:ManagementEvent  where
xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and
fn:number($a/muws2:Situation/muws2:Message)-
fn:number($b/muws2:Situation/muws2:Message)>10  return &lt;x&gt;{$a/muws1:EventId/text()},
{$b/muws1:EventId/text()}&lt;/x&gt;</ss:xquery>
```

```
  </ss:events_analyzer>
  <ss:events_generator>
   <event>Hit rate is declining</event>
   <ss:topic>
    <namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>
    <name>DBHitRateTopic</name>
   </ss:topic>
  </ss:events_generator>
 </ss:sensor>

 <ss:sensor name="WSCALLSENSOR">
  <ss:events_collector>
   <ss:producer uri="http://localhost:8080/PolicyManager/services/PolicyManager">
    <ss:topic>
     <namespace>http://webs2.cs.queensu.ca/AWSEcap/capability/Topics</namespace>
     <name>DSSNESS</name>
    </ss:topic>
   </ss:producer>
  </ss:events_collector>
  <ss:events_analyzer>
   <ss:xquery>xquery declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-
2.xsd";declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ; for $a in
db2-fn:xmlcolumn('WSCALLSENSOR.EVENT')/muws1:ManagementEvent, $b in db2-
fn:xmlcolumn('WSCALLSENSOR.EVENT')/muws1:ManagementEvent  where
xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and
xs:dateTime(fn:string($b/@ReportTime)) &lt; xs:dateTime(fn:string($a/@ReportTime)) +
xdt:dayTimeDuration("PT60S") and fn:number($b/muws2:Situation/muws2:Message)-
fn:number($a/muws2:Situation/muws2:Message)>5  return &lt;x&gt;{$a/muws1:EventId/text()},
{$b/muws1:EventId/text()}&lt;/x&gt;</ss:xquery>
  </ss:events_analyzer>
  <ss:events_generator>
   <event>DSSness is increasing</event>
```

<ss:topic>

<namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>

<name>WSCallTopic</name>

</ss:topic>

</ss:events_generator>

</ss:sensor>

<ss:sensor name="DBSENSOR">

<ss:events_collector>

<ss:producer uri="http://localhost:8080/SensorDomainManager/services/SensorDomainManager">

<ss:topic>

<namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>

<name>DBHitRateTopic</name>

</ss:topic>

</ss:producer>

<ss:producer uri="http://localhost:8080/SensorDomainManager/services/SensorDomainManager">

<ss:topic>

<namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>

<name>WSCallTopic</name>

</ss:topic>

</ss:producer>

</ss:events_collector>

<ss:events_analyzer>

<ss:xquery>xquery declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ; for $a in db2-fn:xmlcolumn('DBSENSOR.EVENT')/muws1:ManagementEvent, $b in db2-fn:xmlcolumn('DBSENSOR.EVENT')/muws1:ManagementEvent where $a/muws2:Situation/muws2:Message/text()='Hit rate is declining' and $b/muws2:Situation/muws2:Message/text()='DSSness is increasing' return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x></ss:xquery>

```
  </ss:events_analyzer>
  <ss:events_generator>
    <event>BP Size + 1000</event>
    <ss:topic>
      <namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>
      <name>DBTopic</name>
    </ss:topic>
  </ss:events_generator>
</ss:sensor>

<ss:sensor name="WSTHROUGHPUTSENSOR">
  <ss:events_collector>
    <ss:producer uri="http://localhost:8080/PolicyManager/services/PolicyManager">
      <ss:topic>
        <namespace>http://webs2.cs.queensu.ca/AWSEcap/capability/Topics</namespace>
        <name>THROUGHPUT</name>
      </ss:topic>
    </ss:producer>
  </ss:events_collector>
  <ss:events_analyzer>
    <ss:xquery>xquery declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-
2.xsd";declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ; for $a in
db2-fn:xmlcolumn('WSTHROUGHPUTSENSOR.EVENT')/muws1:ManagementEvent, $b in db2-
fn:xmlcolumn('WSTHROUGHPUTSENSOR.EVENT')/muws1:ManagementEvent  where
xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and
xs:dateTime(fn:string($b/@ReportTime)) &lt; xs:dateTime(fn:string($a/@ReportTime)) +
xdt:dayTimeDuration("PT30S") and fn:number($b/muws2:Situation/muws2:Message)-
fn:number($a/muws2:Situation/muws2:Message)>10  return &lt;x&gt;{$a/muws1:EventId/text()},
{$b/muws1:EventId/text()}&lt;/x&gt;</ss:xquery>
  </ss:events_analyzer>
  <ss:events_generator>
    <event>Throughput increases</event>
```

<ss:topic>

<namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>

<name>WSThroughputTopic</name>

</ss:topic>

</ss:events_generator>

</ss:sensor>

<ss:sensor name="WSREJECTRATESENSOR">

<ss:events_collector>

<ss:producer uri="http://localhost:8080/PolicyManager/services/PolicyManager">

<ss:topic>

<namespace>http://webs2.cs.queensu.ca/AWSEwscap/capability/Topics</namespace>

<name>REJRATE</name>

</ss:topic>

</ss:producer>

</ss:events_collector>

<ss:events_analyzer>

<ss:xquery>xquery declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ; for $a in db2-fn:xmlcolumn('WSREJECTRATESENSOR.EVENT')/muws1:ManagementEvent, $b in db2-fn:xmlcolumn('WSREJECTRATESENSOR.EVENT')/muws1:ManagementEvent where xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and xs:dateTime(fn:string($b/@ReportTime)) < xs:dateTime(fn:string($a/@ReportTime)) + xdt:dayTimeDuration("PT20S") and fn:number($b/muws2:Situation/muws2:Message)-fn:number($a/muws2:Situation/muws2:Message)>0 return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x></ss:xquery>

</ss:events_analyzer>

<ss:events_generator>

<event>Reject increases</event>

<ss:topic>

<namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>

<name>WSRejectRateTopic</name>

```
      </ss:topic>
    </ss:events_generator>
  </ss:sensor>
  <ss:sensor name="WSSENSOR">
    <ss:events_collector>
      <ss:producer
uri="http://localhost:8080/SensorDomainManager/services/SensorDomainManager">
        <ss:topic>
          <namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>
          <name>WSThroughputTopic</name>
        </ss:topic>
      </ss:producer>
      <ss:producer
uri="http://localhost:8080/SensorDomainManager/services/SensorDomainManager">
        <ss:topic>
          <namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>
          <name>WSRejectRateTopic</name>
        </ss:topic>
      </ss:producer>
    </ss:events_collector>
    <ss:events_analyzer>
      <ss:xquery>xquery declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-
2.xsd";declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ; for $a in
db2-fn:xmlcolumn('WSSENSOR.EVENT')/muws1:ManagementEvent, $b in db2-
fn:xmlcolumn('WSSENSOR.EVENT')/muws1:ManagementEvent where
$a/muws2:Situation/muws2:Message/text()='Throughput increases' and
$b/muws2:Situation/muws2:Message/text()='Reject increases' return
&lt;x&gt;{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}&lt;/x&gt;</ss:xquery>
    </ss:events_analyzer>
    <ss:events_generator>
      <event>Pool Size + 5</event>
      <ss:topic>
```

```
        <namespace>http://awse.queensu.ca/Sensor/capability/Topics</namespace>
        <name>WSTopic</name>
      </ss:topic>
    </ss:events_generator>
  </ss:sensor>
</ss:policy>
</xs:schema>
```

Appendix D

Muse Configuration File

Being WSDN endpoints, Sensor Manager and Policy Manager employ Apache Muse to build Web Service interface for them. We use xml files to configure Muse to expose our endpoints and their capabilities. Every endpoint has one independent Muse configuration file. Therefore, we have two muse.xml files for Policy Manager and Sensor Manager respectively. The following is muse.xml for Policy Manager.

```
<?xml version="1.0" encoding="UTF-8"?>
<muse   xmlns=http://ws.apache.org/muse/descriptor
        xmlns:pfx0="http://webs2.cs.queensu.ca/PolicyManager">
  <router>
   <java-router-class>org.apache.muse.core.routing.SimpleResourceRouter</java-router-class>
   <logging>
    <log-file>/log/muse.log</log-file>
    <log-level>OFF</log-level>
   </logging>
   <persistence>
     <java-persistence-class>org.apache.muse.core.routing.RouterFilePersistence
     </java-persistence-class>
     <persistence-location>router-entries</persistence-location>
   </persistence>
  </router>
  <resource-type use-router-persistence="true">
   <context-path>PolicyManager</context-path>
   <wsdl>
       <wsdl-file>/wsdl/PolicyManager.wsdl</wsdl-file>
       <wsdl-port-type>pfx0:PortType</wsdl-port-type>
   </wsdl>
   <java-id-factory-class>org.apache.muse.core.routing.CounterResourceIdFactory
   </java-id-factory-class>
```

```
<java-resource-class>org.apache.muse.ws.resource.impl.SimpleWsResource
</java-resource-class>
<capability>
 <capability-uri>http://schemas.xmlsoap.org/ws/2004/09/mex</capability-uri>
 <java-capability-class>org.apache.muse.ws.metadata.impl.SimpleMetadataExchange
 </java-capability-class>
</capability>
<capability>
 <capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Get</capability-uri>
 <java-capability-class>
     org.apache.muse.ws.resource.properties.get.impl.SimpleGetCapability
 </java-capability-class>
 </capability>
 <capability>
  <capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Set</capability-uri>
  <java-capability-class>
     org.apache.muse.ws.resource.properties.set.impl.SimpleSetCapability
  </java-capability-class>
</capability>
<capability>
 <capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Query</capability-uri>
 <java-capability-class>
     org.apache.muse.ws.resource.properties.query.impl.SimpleQueryCapability
 </java-capability-class>
</capability>
<capability>
 <capability-uri>http://docs.oasis-open.org/wsdm/muws/capabilities/Identity</capability-uri>
 <java-capability-class>org.apache.muse.ws.dm.muws.impl.SimpleIdentity
</java-capability-class>
</capability>
<capability>
 <capability-uri>
     http://docs.oasis-open.org/wsdm/muws/capabilities/ManageabilityCharacteristics
 </capability-uri>
```

```
    <java-capability-class>
      org.apache.muse.ws.dm.muws.impl.SimpleManageabilityCharacteristics
    </java-capability-class>
  </capability>
  <capability>
    <capability-uri>http://docs.oasis-open.org/wsn/bw-2/NotificationConsumer</capability-uri>
    <java-capability-class> org.apache.muse.ws.notification.impl.SimpleNotificationConsumer
    </java-capability-class>
  </capability>
  <capability>
    <capability-uri>http://docs.oasis-open.org/wsn/bw-2/NotificationProducer</capability-uri>
    <java-capability-class>
      org.eclipse.tptp.wsdm.runtime.capability.SimpleNotificationProducerImpl
    </java-capability-class>
  </capability>
  <capability>
     <capability-uri>http://webs2.cs.queensu.ca/PolicyManager/capability/PublishInterface
      </capability-uri>
      <java-capability-class>ca.queensu.awse.policymanager.EventsPublishCapability
      </java-capability-class>
  </capability>
  <capability>
    <capability-uri>http://webs2.cs.queensu.ca/PolicyManager/capability/Initial</capability-uri>
    <java-capability-class>ca.queensu.awse.policymanager.InitialCapability
    </java-capability-class>
  </capability>
  <init-param>
     <param-name>validate-wsrp-schema</param-name>
     <param-value>false</param-value>
  </init-param>
</resource-type>

<resource-type          xmlns="http://ws.apache.org/muse/descriptor"
                        xmlns:wsntw="http://docs.oasis-open.org/wsn/bw-2">
```

<context-path>SubscriptionManager</context-path>
<wsdl>
<wsdl-file>wsdl/SubscriptionManager.wsdl</wsdl-file>
<wsdl-port-type>wsntw:SubscriptionManager</wsdl-port-type>
</wsdl>
<java-id-factory-class>org.apache.muse.core.routing.CounterResourceIdFactory
</java-id-factory-class>
<java-resource-class>org.apache.muse.ws.resource.impl.SimpleWsResource
</java-resource-class>
<capability>
<capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Get</capability-uri>
<java-capability-class>
org.apache.muse.ws.resource.properties.get.impl.SimpleGetCapability
</java-capability-class>
</capability>
<capability>
<capability-uri>http://docs.oasis-open.org/wsrf/rlw-2/ImmediateResourceTermination
</capability-uri>
<java-capability-class>
org.apache.muse.ws.resource.lifetime.impl.SimpleImmediateTermination
</java-capability-class>
</capability>
<capability>
<capability-uri>http://docs.oasis-open.org/wsrf/rlw-2/ScheduledResourceTermination
</capability-uri>
<java-capability-class>
org.apache.muse.ws.resource.lifetime.impl.SimpleScheduledTermination
</java-capability-class>
</capability>
<capability>
<capability-uri>http://docs.oasis-open.org/wsn/bw-2/SubscriptionManager</capability-uri>
<java-capability-class>org.apache.muse.ws.notification.impl.SimpleSubscriptionManager
</java-capability-class>
<init-param>

```
            <param-name>trace-notifications</param-name>
            <param-value>false</param-value>
          </init-param>
        </capability>
        <init-param>
          <param-name>validate-wsrp-schema</param-name>
          <param-value>false</param-value>
        </init-param>
    </resource-type>
</muse>
```

The following is muse.xml for Sensor Manager.

```
<?xml version="1.0" encoding="UTF-8"?>
<muse   xmlns="http://ws.apache.org/muse/descriptor"
          xmlns:pfx0="http://awse.queensu.ca/SensorManager">
    <router>
     <java-router-class>org.apache.muse.core.routing.SimpleResourceRouter</java-router-class>
      <logging>
        <log-file>/log/muse.log</log-file>
        <log-level>FINE</log-level>
      </logging>
      <persistence>
          <java-persistence-class>org.apache.muse.core.routing.RouterFilePersistence
          </java-persistence-class>
        <persistence-location>router-entries</persistence-location>
      </persistence>
    </router>
    <resource-type use-router-persistence="true">
      <context-path>SensorManager</context-path>
      <wsdl>
        <wsdl-file>/wsdl/SensorManager.wsdl</wsdl-file>
        <wsdl-port-type>pfx0:PortType</wsdl-port-type>
      </wsdl>
```

```
<java-id-factory-class>org.apache.muse.core.routing.CounterResourceIdFactory
</java-id-factory-class>
<java-resource-class>org.apache.muse.ws.resource.impl.SimpleWsResource
</java-resource-class>
<capability>
  <capability-uri>http://schemas.xmlsoap.org/ws/2004/09/mex</capability-uri>
  <java-capability-class>org.apache.muse.ws.metadata.impl.SimpleMetadataExchange
  </java-capability-class>
</capability>
<capability>
  <capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Get</capability-uri>
  <java-capability-class>
    org.apache.muse.ws.resource.properties.get.impl.SimpleGetCapability
    </java-capability-class>
</capability>
<capability>
  <capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Set</capability-uri>
  <java-capability-class>
    org.apache.muse.ws.resource.properties.set.impl.SimpleSetCapability
  </java-capability-class>
</capability>
<capability>
  <capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Query</capability-uri>
  <java-capability-class>
    org.apache.muse.ws.resource.properties.query.impl.SimpleQueryCapability
  </java-capability-class>
</capability>
<capability>
  <capability-uri>http://docs.oasis-open.org/wsdm/muws/capabilities/Identity</capability-uri>
  <java-capability-class>org.apache.muse.ws.dm.muws.impl.SimpleIdentity
  </java-capability-class>
</capability>
<capability>
  <capability-uri>
```

```
    http://docs.oasis-open.org/wsdm/muws/capabilities/ManageabilityCharacteristics
   </capability-uri>
  <java-capability-class>
    org.apache.muse.ws.dm.muws.impl.SimpleManageabilityCharacteristics
  </java-capability-class>
</capability>
<capability>
   <capability-uri>http://docs.oasis-open.org/wsn/bw-2/NotificationConsumer</capability-uri>
   <java-capability-class>
    org.apache.muse.ws.notification.impl.SimpleNotificationConsumer
   </java-capability-class>
</capability>
<capability>
   <capability-uri>http://docs.oasis-open.org/wsn/bw-2/NotificationProducer</capability-uri>
   <java-capability-class>
    org.eclipse.tptp.wsdm.runtime.capability.SimpleNotificationProducerImpl
   </java-capability-class>
</capability>
<capability>
     <capability-uri>http://awse.queensu.ca/SensorManager/capability/ConsumerInterface
     </capability-uri>
     <java-capability-class>
     ca.queensu.awse.sensor.capability.EventsConsumerCapability
     </java-capability-class>
</capability>
<capability>
   <capability-uri>
    http://awse.queensu.ca/SensorManager/capability/PublishInterface
   </capability-uri>
   <java-capability-class>ca.queensu.awse.sensor.capability.EventsPublishCapability
   </java-capability-class>
</capability>
<capability>
   <capability-uri>http://awse.queensu.ca/SensorManager/capability/InitialDomain
```

```
    </capability-uri>
    <java-capability-class>ca.queensu.awse.sensor.capability.InitialDomainCapability
    </java-capability-class>
  </capability>
  <init-param>
    <param-name>validate-wsrp-schema</param-name>
    <param-value>false</param-value>
  </init-param>
</resource-type>

<resource-type          xmlns="http://ws.apache.org/muse/descriptor"
                        xmlns:wsntw="http://docs.oasis-open.org/wsn/bw-2">
  <context-path>SubscriptionManager</context-path>
  <wsdl>
    <wsdl-file>wsdl/SubscriptionManager.wsdl</wsdl-file>
    <wsdl-port-type>wsntw:SubscriptionManager</wsdl-port-type>
  </wsdl>
  <java-id-factory-class>org.apache.muse.core.routing.CounterResourceIdFactory
  </java-id-factory-class>
  <java-resource-class>org.apache.muse.ws.resource.impl.SimpleWsResource
  </java-resource-class>
  <capability>
    <capability-uri>http://docs.oasis-open.org/wsrf/rpw-2/Get</capability-uri>
    <java-capability-class>
      org.apache.muse.ws.resource.properties.get.impl.SimpleGetCapability
    </java-capability-class>
  </capability>
  <capability>
    <capability-uri>http://docs.oasis-open.org/wsrf/rlw-2/ImmediateResourceTermination
    </capability-uri>
    <java-capability-class>
      org.apache.muse.ws.resource.lifetime.impl.SimpleImmediateTermination
    </java-capability-class>
  </capability>
```

```
    <capability>
      <capability-uri>http://docs.oasis-open.org/wsrf/rlw-2/ScheduledResourceTermination
      </capability-uri>
      <java-capability-class>
        org.apache.muse.ws.resource.lifetime.impl.SimpleScheduledTermination
      </java-capability-class>
    </capability>
    <capability>
      <capability-uri>http://docs.oasis-open.org/wsn/bw-2/SubscriptionManager</capability-uri>
      <java-capability-class>org.apache.muse.ws.notification.impl.SimpleSubscriptionManager
      </java-capability-class>
      <init-param>
        <param-name>trace-notifications</param-name>
        <param-value>false</param-value>
      </init-param>
    </capability>
    <init-param>
      <param-name>validate-wsrp-schema</param-name>
      <param-value>false</param-value>
    </init-param>
  </resource-type>

</muse>
```