DESIGN AND IMPLMENTATION OF EVENT BASED

SUBSCRIPTION/NOTIFICATION PARADIGM

FOR DISTRIBUTED ENVIRONMENTS

The members of the Committee approve the masters
thesis of Weera Tanpisuth

Sharma Chakravarthy ____________________
Supervising Professor

Ramez Elmasri ____________________

Leonidas Fegaras ____________________

DESIGN AND IMPLMENTATION OF EVENT_BASED SUBSCRIPTION/NOTIFICATION PARADIGM FOR DISTRIBUTED ENVIRONMENTS

by

WEERA TANPISUTH

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2001

Dedicated to my parents
and my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this challenging topic and providing me great guidance and support through the course of this research.

I would like to thank Dr. Ramez Elmasri and Dr. Leonidas Fegaras for serving on my committee.

I am grateful to Raman Adaikkalavan, Sreekant Thirunagari, and Nishanth Vontela for their invaluable help and advice during the implementation of this work. I also would like to thank Pratyush Mishra for administering the research network. I would like to thank all my friends in the ITLAB for their support and encouragement.

I thank my parents, my mother Rattikorn, my brothers Savee and Nutthapong, for their endless love and constant support throughout my academic career. Without their encouragement and endurance, this work would not have been complete. Last but not least, I thank Nong Porn for extraordinary encouragement and patience.

November 19, 2001

ABSTRACT

DESIGN AND IMPLMENTATION OF EVENT BASED

SUBSCRIPTION/NOTIFICATION PARADIGM

FOR DISTRIBUTED ENVIRONMENTS

Publication No.__________

Weera Tanpisuth, M.S.

The University of Texas at Arlington, 2001

Supervising Professor: Sharma Chakravarthy

A majority of today's applications are distributed in nature and hence there is a need for monitoring and reacting to changes in a distributed application environment. In applications, such as stock tracking, weather alert and forecasting, and network management, timely notification of changes are critical. In the call-driven or demand-based approach, applications that are interested in the monitoring of changes (or specific states) need to take initiative to execute a query to *pull* to infer changes and perform appropriate actions. However, the conventional pull model is inadequate for large-scaled (distributed) environments since it may the window of opportunity if polling is too slow or incur excessive unnecessary system resources if polling is done too frequently. In contrast, the distributed event-based system, based on the event-driven approach (or push), supports timely notification of changes and hence is appropriate for building applications that must monitor and react to changes (in both centralized and distributed environments). A system extended with active capability automatically detects occurrences of the events of interest, and provides timely notification from the producer (notifier) to consumers (subscribers).

This thesis extends the earlier work on Local Event Detector (LED) to provide active capability in a distributed environment. The aim of this thesis is to support event detection across address spaces, provide interface for event registration and notification, and support composite event detection. This thesis discusses the design and implementation of the distributed event-based system, which is called Global Event Detector (GED). The design is based on the notify/subscribe paradigm, and uses the Event-Condition-Action (ECA) rules to support active capability: when an event is detected, a condition is evaluated and an action is executed if the condition is satisfied. This thesis also summarizes the differences between an earlier C++ version and the current Java version in terms of functionality and advantages.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A majority of applications today are distributed in nature and hence there is a need for monitoring and reacting to changes in a distributed application environment. For example, Element Management System (EMS), used to monitor and manage a network of distributed components, such as controller cards in Multi Service Fabric (MSF) and links between signaling points (SP) requires a change detection and notification mechanism. These network components are subject to failures in many ways. For instance, the links are broken or the cards are overloaded. When an unpredictable and undesirable problem occurs, the application operators prefer to be notified rather than executing a query to pull the status from a massive repository. For these and other similar application environments, active capability offers a promising approach to detect events and respond with user-defined actions if appropriate conditions are satisfied.

The early work on the Local Event Detector (LED), and other active notification systems such as Observer/Observable from Java and Producer-Bus-Consumer from InfoBus [1] are limited to their own address space. That is, they are well suited for stand-alone applications. In CORBA [2] (Common Object Request Broker Architecture), the event service decouples the communication between suppliers and consumers. Suppliers produce event data and consumers process event data through the standard CORBA request. Even though the CORBA's event service provides the event notification in distributed environments, the notion of composite event detection has not been addressed.

This thesis extends the earlier work that provides support for events and rules in Java applications in a seamless manner [3] within an application environment or address space. Our focus is on the global event specification and primitive and composite event detection in a distributed environment by using event-based subscription/notification paradigm. The subscription is completely localized in the sense that the subscriber does not have to even worry about the presence of the event producer. As and when the producer comes on-line, event subscription is made by a mediator on behalf of the subscriber and notification is also handled by the same mediator. This thesis discusses the architecture, design, and implementation of the global event detection system.

The Global Event Detector (GED) is a server based on the notification/subscription paradigm. All message passing is done in a demand-driven mode. That is, no messages are sent to the server unless there is a consumer for that message. The server receives an event detection request from a consumer application and forwards it to the corresponding producer only when it registers with the server. When the event of interest defined in the producer application occurs, the producer application notifies the occurrence of event to the server. The server not only forwards the occurrence of the event to the corresponding consumers, but it is also responsible for detecting any composite event based on that event. Our implementation of the GED uses the ECA (Event-Condition-Action) rule paradigm in order to support active capability in a distributed environment. According to the ECA rule paradigm, whenever the event occurs, the condition defined in the rule (for that event) is evaluated and the corresponding action is performed if applicable. The outline of this thesis is as follows. Chapter 2 reviews the research area of active database systems and some of the related work. Chapter 3 discusses the semantics of events and rules in a distributed environment. Chapter 4 summarizes the design and implementation of the local event detector (LED). Chapter 5 discusses the architecture alternatives for the distributed extension, describes the communication mechanisms, and explains the basic event detection mechanism. Chapter 6 describes the implementation of global event detection and the server components. Chapter 7 concludes the thesis with a summary and some suggestion for future work.

CHAPTER 2

RELATED WORK

This chapter reviews the related work in the area of event-based notification. We will discuss the research from Object Management Group (OMG), Microsoft Corporation, Cambridge Computer Laboratory, and Global Event Detector from the Sentinel group.

2.1 Common Object Request Broker Architecture

Based on publish/subscribe paradigm, event service is one of the services from Common Object Request Broker Architecture (CORBA). [2] [4] It defines three roles (supplier, consumer, and event channel). The suppliers and consumers are decoupled, and transparent from each other. Suppliers can push data to consumers through the event channel. Likewise, consumers can use event channel to pull the data from suppliers. The event channel works as a mediator between consumers and suppliers. The event channel inte4rface can be used for adding consumers, adding suppliers, and also for destroying the channel.

There are four models of component collaboration in the event service architecture:Push Model, Pull Model, Hybrid Push/Pull Model, and Hybrid Pull/Push Model. The Push model allows suppliers to initiate the transfer of event data to consumers. In the pull model, the consumers request the event data from suppliers through event channel. The Hybrid Push/Pull model allows both consumers and producers to initiate the transfer of event data. The event channel plays the role of a passive mediator. The active consumers can request data via the event channel in which the active supplier pushes the data. The Hybrid Pull/Push model, in contrast to the Hybrid Push/Pull model, allows the active event channel to pull the data from suppliers and push them to consumers.

Although the symmetry provided by the COBRA event service is well designed, it is not possible for consumers to subscribe only to events, which are of interest. Each event that is sent from each producer to the event channel is delivered to all the registered consumers. This requires consumers to filter out event data that is not of interest. It involves additional overhead for the consumers. This is likely to increase the network utilization and cause clogged network traffic. In addition, there is no notion of composite events. Thus, it does not support advanced event handling.

2.2 COM+ Events Model

COM+ [5] is an event model from Microsoft. The push model is supported as event notification model. It defines three roles (publisher, subscriber, and COM+ events). Unlike CORBA's event service, it only supports push model. It allows the subscribers to request event notifications from a particular producer. The subscribers and publishers are loosely coupled. Acting as a mediator, COM+ Events system stores event information from different publishers in an event store, and allows subscribers to select subscriptions, which are of interest and register them with the COM+ event service. When publishers fire events; the mediator looks in the event store, and finds the subscribers that have registered to the particular subscriptions, and notify subscribers. However, COM+ Events model does not support distributed applications. In addition, there is no notion of composite events.

2.3 Schwiderski Dissertation

This dissertation [6] presents an approach to event-driven monitoring of the behavior of distributed systems. It defines an event specification language for primitive and composite events that are used in a distributed system. Five event operators (conjunction, disjunction, sequence, iteration, and negation) are introduced to construct composite events that are applicable to events at either local or remote sites. In addition, formal semantics of primitive and composite event are defined to identify when and where an event occurs. Furthermore, the detection algorithm and the architecture of the global event detection are presented. The local and global event detections are distributed. Thus, each site has both a local event detector and a global event detector to support event detection.

This design, however, does not directly address issues of event registrations. It uses a configuration file to define the global event trees in order to construct the event graph. In addition, the prototype implementation is based on a conventional RPC system. But RPC does not support object communication. Hence, it is not suitable for application using the object-oriented paradigm. It uses Modula-3 network object, which is a distributed programming system. Thus, it lacks system interoperability. Furthermore, it doesn't handle the complex event parameters and contexts. It doesn't address how to handle parameters that are associated with the composite event, and the evaluation of global event trees is presented in only the chronicle context.

2.4 Sentinel Global Event Detector

The Sentinel project [7] addresses design and implementation of a global event detector in the C++ environment. The major contributions of this project are the extension of the event specification language (termed Snoop) and Snoop Preprocessor (SPP), and the implementation of a global event detector. Snoop used to define ECA rules is extended to express event semantics for the distributed environments. A Snoop Preprocessor transforms the event and rule definitions written in Snoop to the conventional C++ programming code. It also generates the global event specification file, which contains the information used by the server for detecting the event defined outside of a local application. Based on this information, the global event detector constructs the event graph for the detection of primitive and composite global events.

This implementation presents an approach to overcome the shortcomings of CORBA mentioned above. However, there are still some limitations introduced by C++ environment in providing active capability.

- **Portability**: Implemented in C++, the Global Event Detection uses system calls to the underlying operating system to support temporal event detection. This implementation, thus, is difficult to port and use on a different platform.
- **Object Oriented Paradigm**: The Global Event Detector relies on a conventional RPC communication. This technique is not suitable for applications using the object-oriented paradigm since the communication is restricted to the primitive data type such as integer, float, and double. In order to support object communication, it requires tedious and enormous work to marshal parameters at the client and unmarshal at the server. Furthermore, the condition and action parts of rules were modeled as C functions, which are typically used in procedural programming paradigm. Thus this compromises object-oriented paradigm
- **Resource Utilization**: Each application independently uses the Snoop Preprocessor to transform the ECA rules into conventional C++ code, and generate event specifications in a flat file. Each application reads this information at run time and sends it to the server to construct the event graph for detecting both primitive and composite events. Since the structure of the event graph is identified by each application at the pre-compilation time and sent separately to the server, two applications cannot share the same event graph even though they are interested in the same event supplied by the same producer. When the event occurs, producers have to send the duplicate

notifications to the server in order to notify each corresponding node. This is likely to increase resource utilization

CHAPTER 3

SEMANTICS OF EVENTS

In an active database system, the expressiveness of event specification is important for modeling complex applications. In a single application scenario, all the occurrences of events (primitive or composite) relate to one application (or address space). On the other hand, the occurrences of events in a distributed application scenario relates to more than one application (or address space). In this chapter, event specification for primitive and composite are discussed. The extensions to support specification and detection of events across address spaces (termed global event detection) are needed to support a distributed application environment. This chapter describes the types of events needed in a distributed environment. This chapter also discusses event operators used in composite event expressions, associated semantics, and the notion of parameter contexts.

3.1 Primitive Event

An event is an occurrence of interest at a specific point in time. Primitive events are the elementary occurrences and are classified into database, temporal, and explicit events. Database events are associated with the manipulation of data such as the creation, deletion, or insertion that are executed over a period of time. Event modifiers (begin and end) were introduced to transform operations that take an interval into an event. In other words, the event modifiers (begin and end) are used to map the logical events at the conceptual level to physical events. The begin event modifier denotes the starting point of a database event and the end event modifier denotes the ending point. Temporal events correspond to absolute and relative temporal events. The absolute temporal event is an event associated with an absolute value of time. For example, 4 P.M. on September 11, 2001 is an absolute event. The relative temporal event is an event corresponding to a specific point on the time line, which is an offset from another time point (specified either as absolute or as an event). Explicit (also termed abstract) events are explicitly defined in the application, but their occurrences are either detected outside of the application and conveyed to the application or the application explicitly raises those events.

In a distributed environment, primitive events can be broadly classified into two types: local primitive and global primitive.

3.1.1 Local Primitive Event

Local primitive events are the database (or domain specific), temporal, or explicit events that are defined in an application and are detected (or raised) in the same site as that of application. [7] An event is defined in terms of a name, an event modifier, a method signature (method whose execution raises the event), a class (with which the event is associated) and parameters (arguments of the method plus the instance on which the method is executed). The underlying event detection mechanism (the local event detector) uses this information to determine the occurrence of an event. The parameters of the event are gathered at the time of occurrence and are preserved for later use by condition and action components of an ECA (event-condition-action) rule. Here is an example of local primitive event. A meteorologist in Arlington weather center likes to know when the temperature reaches 100 degrees Fahrenheit. This event is defined in the Arlington weather application on the update temperature method along with the condition that checks for the new temperature as to whether it has reached 100 degree Fahrenheit and an action that informs the meteorologist about this happening. Whenever the event occurs (that is, the update temperature method is executed), the system will check for the condition and if it evaluates to true, the action is executed notifying to the meteorologist by mail (in this case).

3.1.2 Global Primitive Event

Global primitive events are those the events that are defined and detected outside the current application and are subscribed to by the current application in the context of a distributed application scenario. In other words, an application considers an event (either a primitive or composite) defined outside of that application as a global primitive event. Since the remote applications are autonomous, only an event name is not sufficient to distinguish events form different application as two applications may have the same event name. In order to avoid any potential ambiguity, additional information such as an application name and a machine name are used to refer to a primitive global event. The following are examples of composite global events. The Arlington meteorologist wants to know when the humidity in Austin goes above 40 percent. This local primitive event is defined and detected in the application located in Austin; however, the occurrence is forwarded to the application in Arlington. Another example is that the Arlington meteorologist might be interested when the temperature in Austin is below 50 degrees Fahrenheit and is raining. As described in the next section, the local composite event in Austin (corresponding to the composite event temp < 50 AND raining) is considered to be a global primitive event for the application running at Arlington.

3.2 Composite Events

A composite event is an event that is composed of primitive events and/or other composite events by applying Snoop [8] [9] [10] event operators such as OR, AND, SEQUENCE, and NOT. These operators will be discussed later in this chapter. In order words, the constituent events of the composite event can be primitive events and/or composite events.

In the context of a distributed application, a composite event can be classified into two types: local composite event and global composite event.

3.2.1 Local Composite Event

Local composite events are composed of local primitive events and/or other local composite events with event operators. [11] The local composite event and all its constituent events are locally defined and detected in one site. The occurrence can be detected by using a local detection mechanism. For example, the meteorologist in Arlington weather center wants to know when the temperature is over 100 degrees Fahrenheit and the humidity is less than 40 percent (in Arlington). All the constituent events are defined and detected in the Arlington weather application.

3.2.2 Global Composite Event

Global composite events are events whose constituent events are related to the event occurrences from many sites including the local site. They can be composed of local primitives, local composite events, global primitive events and/or other global composite events by applying one or more of the event operators; however, one of the constituent events must be a global event.

3.3 Event Operators

The even operators are used to construct composite events. Each of these event operators and its semantics are described briefly in the following section. The upper case letter E, which represents an event type, is a function from the time domain on the Boolean values. The function is given by

$$E\,(t) = \begin{cases} \text{True if an event type E occurs at time point t} \\ \text{False otherwise} \end{cases}$$

- **Disjunction: OR (∇)**

Disjunction of two events E_1 and E_2, denoted by $E_1 \nabla E_2$ is applied when either E_1 occurs or E_2 occurs. Formally,

$$(E_1 \nabla E_2)(t) = E_1(t) \nabla E_2(t)$$

- **Conjunction: AND (Δ)**

Conjunction of two events E_1 and E_2, denoted by $E_1 \Delta E_2$ is applied when E_1 occurs and E_2 occurs in any arbitrary order. Formally,

$$(E_1 \Delta E_2)(t) = (E_1(t1) \Delta E_2(t)) \nabla ((E_1(t) \Delta E_2(t1))$$
$$\text{and } t1 \leq t$$

- **Sequence (;)**

The sequence of two events E_1 and E_2, denoted by $E_1 ; E_2$ occurs when E_1 happens before E_2. The timestamp of occurrence of E_1 is less the timestamp of occurrence E_2. Formally,

$$(E_1; E_1)(t) = E_1(t1) \Delta E_2(t) \text{ and } t1 < t$$

- **Negation: NOT (¬)**

The *not* operator, denoted by $\neg(E_2)[E_1, E_3]$ is applied when there is no occurrence of E_2 in the closed interval formed by E_1 and E_3. Formally,

$$\neg(E_2)[E_1, E_3](t) = (E_1(t1) \Delta \sim E_2(t2) \Delta E_3(t))$$
$$\text{and } t1 \leq t2 \leq t$$

- **Aperiodic Operators (A)**

The non-cumulative aperiodic operator, denoted by $A(E_1, E_2, E_3)$ is used to express the occurrence of an aperiodic event in the half-open interval formed by two arbitrary events. The A event occurs each time when E_2 occurs during the half-open interval defined $E_1$and E_3. The number of occurrence of A event is proportional to the occurrence of E_2 during the interval. Formally,

$$A(E_1, E_2, E_3)(t) = E_1(t1) \Delta \sim E_3(t2) \Delta E_2(t))$$
$$\text{and } (t1 < t2 \leq t \text{ or } t1 \leq t2 < t)$$

- **Aperiodic Star Operators (A*)**

The cumulative aperiodic operator, denoted by A* (E_1, E_2, E_3) is similar to the non-cumulative aperiodic operator. It A* event is signaled only once when E_3 occurs rather than detected the event every time the event E_2 occurs. However, the occurrences of E_2 are accumulated each time when E_2 occurs during the half –open interval formed by E_1 and E_3. Formally,

$$A^*(E_1, E_2, E_3)\ (t) = (E_1\ (t1)\ \Delta\ E_3\ (t)) \text{ and } t1 < t$$

- **Periodic Operator (P)**

The periodic operator, denoted by P (E_1, [t], E_3) is used to express a periodic event that repeats itself within a constant and finite amount of time. The event P is signaled for every amount of time t in the half-open interval (E_1, E_3]. Formally,

$$P\ (E_1, [TI], E_3)\ (t) \ = \ (E_1(t1)\ \Delta \sim E_3(t2))$$

$$\text{and } t1 < t2 \text{ and } t1 + x * TI = t \text{ for some } 0 < x < t \text{ and } t2 \leq t$$

where TI is a time specification.

- **Periodic Star Operator (P*)**

The periodic star operator denoted by P* (E_1, [t], E_3) is a cumulative variant of P. The occurrence Similar to aperiodic star context, the P* event is signaled only once when E_3 occur. Formally,

$$P^*(E_1, [TI], E_3)\ (t) \ = \ (E_1(t^1)\ \Delta\ E_3(t))$$

$$\text{and } t^1 \leq t$$

- **Plus (+)**

The plus operator denoted by E_1+ [T] is applied when T time units are elapsed after E_1 occurs.

3.4 Parameter Context

Four parameter contexts—recent, chronicle, continuous, and cumulative—were introduced to provide a mechanism for capturing meaningful application semantics and reduce the space and computation overhead for the detection of composite events using the semantics described above. [9] [11] The contexts are defined by using the notions of initiator and terminator for events. An event that initiates the occurrence of a composite event is termed the initiator of the composite event. An event complete the detection of composite event is denoted the terminator of the composite event. For example, a composite event (E1 Δ E2 Δ E3) has E1 as an initiator and E3 as a terminator.

- **Recent**:

In the recent context, only the most recent occurrence of the initiator (when there are multiple instances of the same event) for any event that has started the detection of that event is used. When the event occurs, all the occurrences of events, that are used in the parameter relation and cannot be initiators of that event in the future, are deleted. In this context, not all occurrences of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event will continue to initiate new event occurrences until a new initiator occurs. This recent context is suitable for events that happen at a fast rate and the multiple occurrences of the same event only refine the previous value.

- **Chronicle**:

In the chronicle context, the initiator-terminator pair is unique for an event occurrence. The oldest initiator is paired with the oldest terminator for each event. When event occurs, the occurrences of the events are deleted. The event occurrence can be used at most once for computing the parameters of the composite event. This context is useful when the different types of events have an established relationship between their occurrences.

- **Continuous**:

In the continuous context, each initiator of an event starts a separate detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and terminator are discarded after an event is detected. This context is fit for tracking trends of interest along a moving time window.

- **Cumulative**:

  In the cumulative context, all occurrences of an event type are accumulated as instances of that event until the event is detected. When the event occurs, all the occurrences that are used for detecting are discarded.

3.5 Coupling Modes

In early systems such as POSTGRES [12], condition evaluation and action execution were done immediately after the event was detected. However, in some situations this is too restrictive. For integrity checks, condition evaluation and action execution need to be done at the end of a transaction before it commits. Coupling modes were introduced to specify a relative point in time where condition evaluation and action execution should take place after the event is detected, with the constraint that the action will be performed only when the condition is satisfied. There are three coupling modes:

- **Immediate**:

  When an event is detected, the transaction is suspended, and the condition associated with the event is evaluated immediately. If the condition evaluates to true, the action is executed. The execution of the triggering transaction is continued when the condition evaluation and action execution are completed.

- **Deferred**:

  The triggering transaction is continued after an event is detected. Condition evaluation and action execution are done at the end of the triggering transaction before it commits.

- **Detached (or decoupled)**:

  Condition evaluation and action execution are done in a separate transaction (or triggered transaction) from the triggering transaction. The detached mode can be classified into two types (totally independent and causally dependent). When two transactions are totally independent, the triggered transaction is executed regardless of whether the triggering transaction commits or aborts. On the other hands, the triggered transaction can commit only after the triggering transaction commits for the causally dependent mode.

IMPLEMENTATION OF PRIMTIVE EVENTS

In this chapter, we discuss, in detail, the issues concerning the creation and detection of primitive events. The reason we need to know about the occurrence of primitive events outside the scope of the SQL Server is to primarily aid the Java LED in the detection of Composite events. We have to remember here that; we are using another layer of software over the SQL Server in order to enhance the trigger capability of the SQL Server. This layer of Software is what is called *The Generalized Mediator Agent* or *The ECA Agent*. This ECA Agent uses the trigger mechanism of the underlying SQL Server as the basis and builds upon it, in order to enhance the triggering capability and turn the system into a true Active DBMS.

CHAPTER 4

SUMMARY OF JAVA LOCAL EVENT DECTOR

This chapter summarizes the design of java local event detector (LED) and its functionality. The local event detector, based on the ECA rule paradigm, provides flexible and expressive event semantics in order to support active capability useful to various kinds of applications, including relational and object-oriented database systems. The LED has been used to develop a agent/mediator that work with various commercial Relational DBMSs (such as Oracle, DB2 and Sybase). [13] [14] The local event detector is well suited for monitoring complex changes with in an application. In addition to the overall architecture, this chapter discusses various data structures used in local event detection.

4.1 Event Specification Interfaces

In a centralized system, the local event detector provides active capability in an application by detecting the occurrence of local primitive and local composite events defined as part of the application. The application interacts with the local event detector through a set of interfaces (API).

This following section describes the steps for using the event detector. First, the application has to initialize an agent by invoking the *initializeECAAgent( )* or *initializeECAAgent(String agentName*) method. When the *initializeECAAgent( )* method is invoked for the first time, the system initiates a new ECA agent (called the *defaultECAAgent when invoked without a parameter)*; otherwise, it will return the existing defaultECAAgent to the application. The application can initialize multiple agents by using *initializeECAAgent (String agentName)*. Each agent is responsible for monitoring

events defined in that agent and performing appropriate actions. After initialization, the application can start defining events (both primitive and composite), and rules. The APIs for initializing an agent are shown below.
Initialize ECAAgent API :

initializeECAAgent ()

initializeECAAgent (String agentName)

For defining a primitive event, the application basically specifies a name for the primitive event, a name of the class in which the method associated with the event is defined, the event modifier (begin or end), and the complete method signature. The application uses the initialized ECAAgent to invoke with the *createPrimitiveEvent* method to define the primitive event. When an event is defined using the API shown below, the associated event handle is returned. This handle is used for defining the composite event, storing the parameter of the event as well as, signaling the method invocation of event to the detector.

There are mainly two APIs for defining a composite event. First API is used for defining a binary Snoop operator such as AND, OR, and SEQUENCE with two constituent events, which can be either primitive events or other composite events. The other API is for defining a composite event that has three constituent events. The examples of a ternary operator are NOT, PERODIC, and APERIORDIC. To define a composite event, the application specifies the operator type and event handles obtained from previous declarations as described above. Similar to defining a primitive event, the application calls the *createCompositeEvent* method to define a composite event in a specific ECA agent. The following interfaces is used for defining an event

Create Primitive Event API

createPrimitiveEvent (String eventName, String className,

EventModifier modifier, String methodSignature)

Create Composite Event API

createCompositeEvent (EventType operator, String eventName,

EventHandle ehOne, EventHandle ehTwo)

createCompositeEvent (EventType operator, String eventName, EventHandle ehOne, EventHandle ehTwo, EventHandle ehThree)

The rule definition basically consists of an event handle (corresponding to an event) with which the rule is associated, a condition method name, and an action method name. To specify a rule, the application invokes *createRule (EventHandle eh, String ruleName, String condMethod, String actionMethod)*. The condition and action are defined as methods in the associated class. The rule is fired when the event occurs and the condition is evaluated. If the condition is satisfied, the action will be performed.

In some cases, we can define an event and a rule specification in a separate class. To define a rule that has a condition and an action defined in another class, the application specifies the class name (in which the method associated with the rule is defined) in the *condMethod* and *actionMethod*. The format is as follows:

condMethod = “PackageName”+”.”+ClassName”+”.”+”MethodName”

actionMethod = “PackageName”+”.’+”ClassName”+”.”+”MethodName”

For a class-level rule, the application invokes the same API as: *createRule (EventHandle eh, String ruleName, String condMethod, String actionMethod)*. To apply the rule on the specific instance of the class in which condition and action methods are defined, the application needs to add a target object , termed a *targetInstance*, in the a rule. The APIs for defining a rule are as follows.

Create Rule API

createRule (String ruleName, EventHandle eh, String ruleName, String condMethod, String actionMethod)

createRule (Object targetInstance, String ruleName, EventHandle eh, String ruleName, String condMethod, String actionMethod)

In the body of the method defined as a primitive event, arguments of that method need to be inserted into the parameter list using the even handle before raising the event. The application can insert any primitive data type or object through *insert( )* API shown below. As described in the previous chapter, these parameters will be used for checking conditions and performing actions when the rule is fired. At the time a primitive event is raised, the application signals the event detector through a method so that the detector can trap the occurrence of the primitive event. Typically, the application calls raiseBeginEvent (EventHandle [] eventHandleArray, Object instance) or raiseEndEvent (EventHandle [] eventHandleArray, Object instance) inside the method. The API used depends on whether the event is raised at the beginning or at the end of the method. The application uses event name to retrieve the vector of event handles by calling *getEventHandles (String eventName)* API in the *ECAAgent* class.

Insert Parameter API

insert(EventHandle[] eventHandleArray, String varName, long longValue)

insert(EventHandle[] eventHandleArray, String varName, float floatValue)

insert(EventHandle[] eventHandleArray, String varName, Object object)

Raise Event API

raiseBeginEvent(EventHandle[] eventHandleArray, Object instance)

raiseEndEvent(EventHandle[] eventHandleArray, Object instance)

4.2 Event Detection

LED uses on an event graph for detecting composite events as shown in figure 4.1. Each node in the event graph represents either a primitive event or a composite event defined in the application by using above interfaces. Primitive event nodes are leaf nodes from which composite event nodes are constructed. Composite event node can also be constructed from primitive events and/or other composite events. The primitive event node contains an instance-based multiple rule list and an event subscriber list, while the composite event node contains only one rule subscriber list and an event subscriber list. An instance-rule list is a collection of rule subscriber lists for classes and instances. The rule subscriber list and event subscriber list keep the associated rules and composite events respectively. The *eventSignaturesEventNodes* and *eventNamesEventNodes* hash tables provide references to the primitive event nodes. When a primitive event occurs, it is inserted to the leaf node and the occurrence is propagated to the internal nodes similar to a data-flow computation. The occurrence is propagated by means of an event table, which is described in the next section.

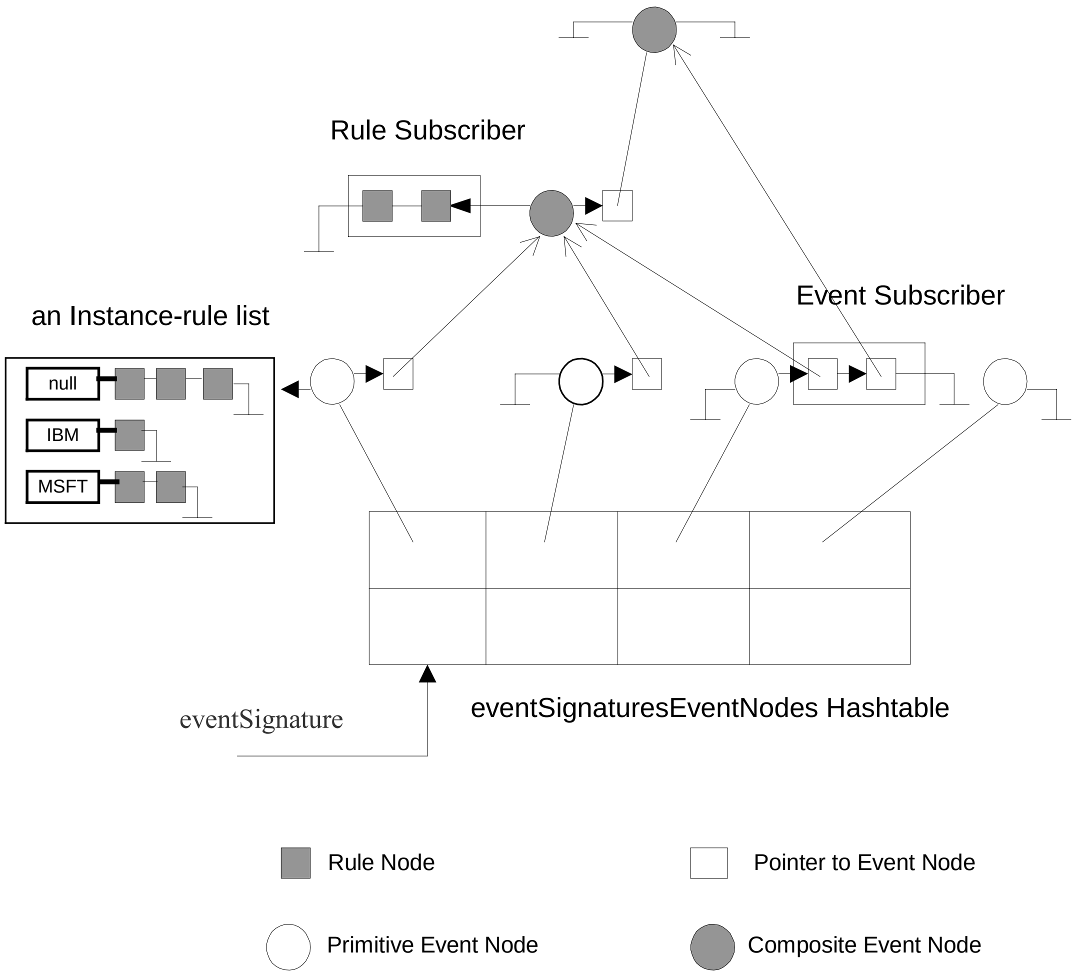


Figure 4.1. Event graph for detecting primitive and composite events.

Figure 4.2 shows an event table that stores all the key information including event occurrences, timestamp, a list of parameter lists, and context information used in detecting an event. The event table is also passed to the condition and action methods for extracting appropriate parameters for usage within condition and actions methods. An event table, termed a PCTable, consists of a set of event entries. Each event entry holds four-bit context information (Recent, Chronicle, Continuous, and Cumulative) and a list of parameter lists. The parameter list denotes the occurrence of a primitive event, whilst the list of parameter lists denotes the occurrence of a composite event. The number of parameter lists in the list will be proportional to the number of constituent primitive events. The parameter list is used to store information when the event is raised, and to retrieve the data when the event is detected. The data coming form the parameter list is used for checking condition and performing action according to the rule definition.

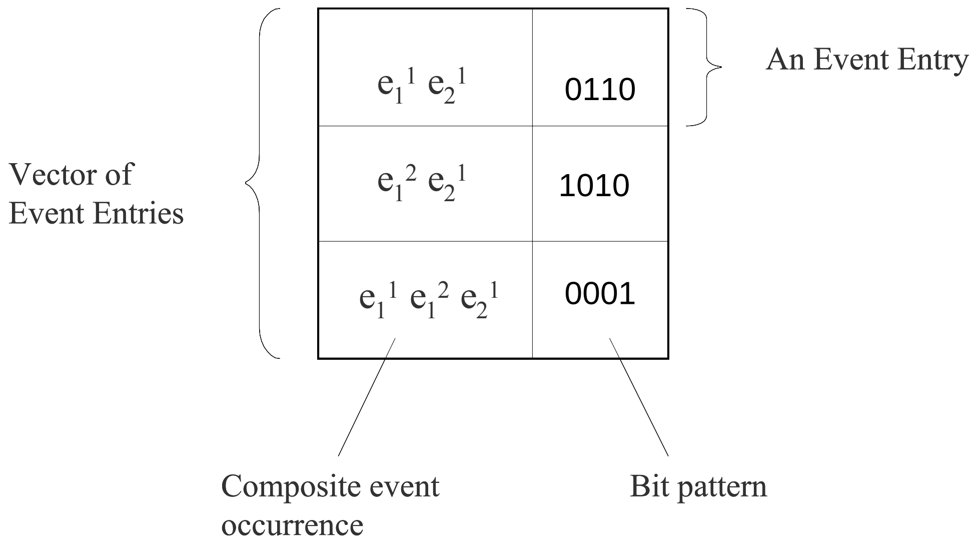


Figure 4.2. An event table represents a set of event occurrences.

The rule subscriber list consists of rule nodes, each of which defines a rule. The rule node stores the rule name, references to context in which the rule should be executed, condition method name and action method name. The rules in the rule subscriber list are triggered when the associated event occurs in that particular context. Whenever the rule is fired, the rule thread that contains the rule definition is instantiated. And it will be inserted into the rule scheduler. Once the rule is executed, the condition will be evaluated to determine whether the action should be performed or not.

4.3 Overview of local event detector architecture

The main components of the local event detection system are event, rule, ECA agent, rule scheduler, and event detector thread

The event nodes are used to represent defined events. As mentioned in chapter 3, the events of interest are classified into two main groups (primitive and composite) for the local event detection. Therefore, an Event class is derived into two subclasses, which are a Primitive event and a Composite event class. And the composite event is derived into subclasses, which correspond with the event operators such as AND, OR, SEQUENCE. For ease of use, the application is provided with an event handle for each primitive event (or node). The application can obtain the event handle by giving an event name. . The EventHandle object stores a method signature, a class name, and a list of parameter lists associated with that event. It is used for various tasks by the application. The user uses the EventHandle object when creating the composite event or inserting the parameters. The purpose of the event handle is to reduce the amount of information the user needs to keep track of. The event handle allows one to obtain all other information by remembering the event name. Figure 4.3 depicts the event class hierarchy.

A rule consists of a condition and an action. The condition and action are implemented as methods of a class in Java. Since LED uses Java reflection, the rule object needs to know the name of a method to reference back to the condition and action method. In addition, it keeps the context information for composite event detection. Whenever the rule is triggered, the *RuleThread* object is instantiated if rule scheduler is turned on. Then, this instance is inserted into the rule queue and executed by rule scheduler. Figure 4.4 shows the relationship between *Rule*, *RuleThread*, and *Event* class.

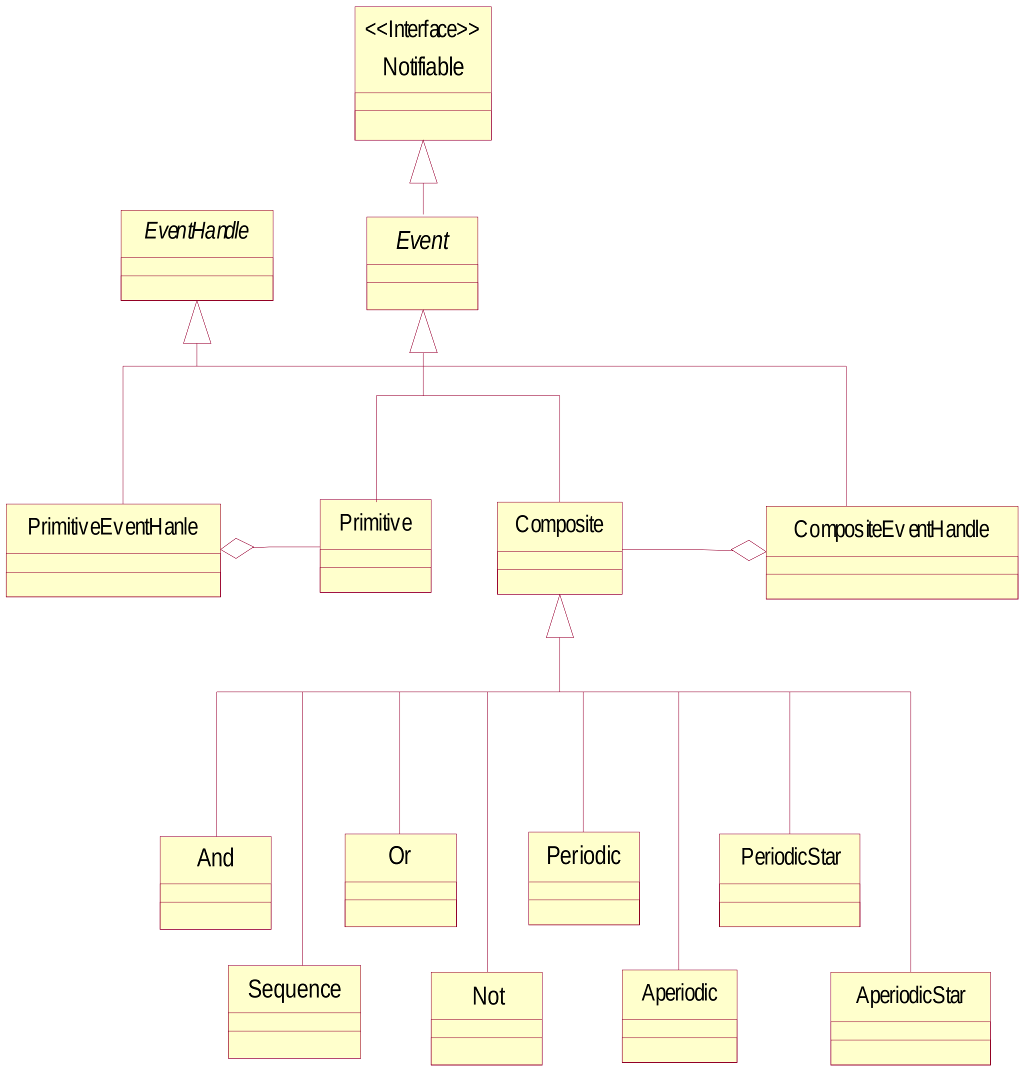


Figure 4.3. Event Class Hierarchy.

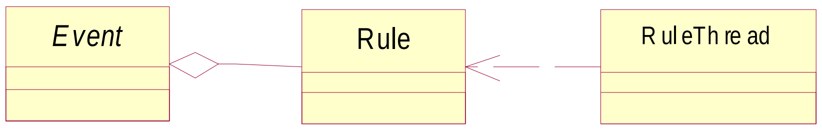


Figure 4.4. Rule Class Diagram.

The ECAAgent is responsible for monitoring the events of interest in an application. It provides interface to define events and rules, insert parameters using the event handle, and signals an event occurrence to the event detector. The application can name the agent at the initialization time. When the agent is initialized without a specific name, the *defaultECAAgent* is created. After initialization, this default agent can be referred to at any time by using the static method, *ECAAgent.getDefaultECAAgent( )*. The ECAAgent class maintains two hash tables. The *eventNameEventNodes* and *eventSignaturesEventNodes* hash tables store a mapping to the event nodes. These two hash tables work as a registry, which is internally used for referring back the event node when the event occurs. Since the application thread is separated from the event detector, it works in conjunction with an event detector thread (*LEDThread*) through a buffer (*NotifyBuffer*) as shown in figure 4.5

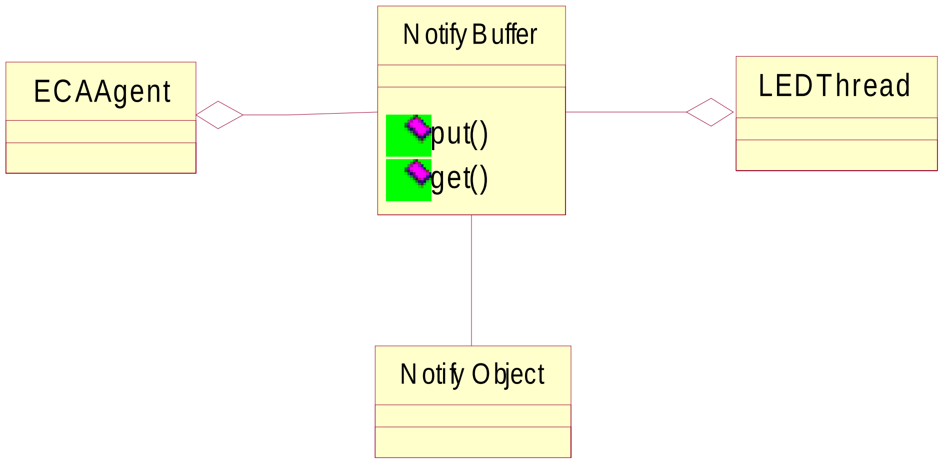


Figure 4.5. Notify Buffer and Notify Object Class Diagram.

The event detector thread is responsible for detecting events and firing rules. Whenever a primitive event occurs, all the relevant information about its occurrence is wrapped into an object called *NotifyObject*. This *NotifyObject* is put into the buffer, and is processed by the *LEDThread*. Running in an infinite loop, the event detector thread keeps fetching NotifyObjects, notifying occurrences of events, propagating the parameters to the internal nodes of the event graph, and firing the associated rules. The application thread that raised an event is synchronized according the immediate rule firing semantics.

CHAPTER 5

DESIGN OF THE GLOBAL EVENT DECTOR

This chapter describes the design of the global event detector. New interfaces (APIs) introduced to accommodate event specification for a distributed environment are explained. In addition, alternative architectures and communication mechanisms along with their advantages and disadvantages will be discussed. One of our goals is to minimize the computation and communication costs by determining an appropriate location where a global event is managed. All the requirements of global event detection are addressed in this chapter. We also summarize Remote Method Invocation (RMI) and object serialization mechanisms used for communication.

5.1 Global Event Specification

Two global event types are supported in our development: global primitive and composite event. As defined in the chapter on semantics of events, global primitive event is an event that is defined and detected outside of current or local application. Event name, application name, and host name are used to compose a global primitive event specification. The event detection and network communication details are transparent to users. The API used for the global primitive event specification is as follows:

Create Global Primitive Event API

```
createPrimitiveEvent (String consEventName, String className,
                      String prodEventName, String appName, String machName)
```

consEventName is the event name defined by a current or local application,and can be used to reference an event handler. className is a name of the class in which the current or local application subscribes to a global primitive event. prodEventName is the event name that is defined in the other application where this event is detected. appName and machName denote the name of the application and machine name where this global event is defined.

Global composite event is an event that is composed by event operators and at least on e of its constituent events is a global event. The global composite event specification is identical to the local composite event specification. The internal mechanism will determine the type of composite event (local or global) and the site of a global composite event detection at run-time. The details of composite event detection site are described later in this chapter. The APIs for defining a composite event are as follows:

Create Composite Event API

```
createCompositeEvent  (EventType operator, String eventName,
                       EventHandle ehOne, EventHandle ehTwo)
createCompositeEvent  (EventType operator, String eventName, EventHandle ehOne,
                       EventHandle ehTwo, EventHandle ehThree)
```

5.2 Goals of Global Event Detection

Applications running on different sites and possibly using heterogeneous platforms constitute a distributed application environment. Any monitoring mechanism for a distributed environment should provide a mechanism for subscribing to changes in other applications as well as changes that span, multiple applications. Global event detection is introduced as a mechanism to accomplish the above. This entails enhancing the local event detector to communicate with a global event detector as well as sending appropriate events from one address space to another. Our approach also endeavors to minimize the overhead associated with message communication. The communication alternatives are taken into account to enhance the system performance.

5.3 Architecture Alternatives

There are two main approaches to detect global events in a distributed environment.

5.3.1 Distribute global event detection among applications

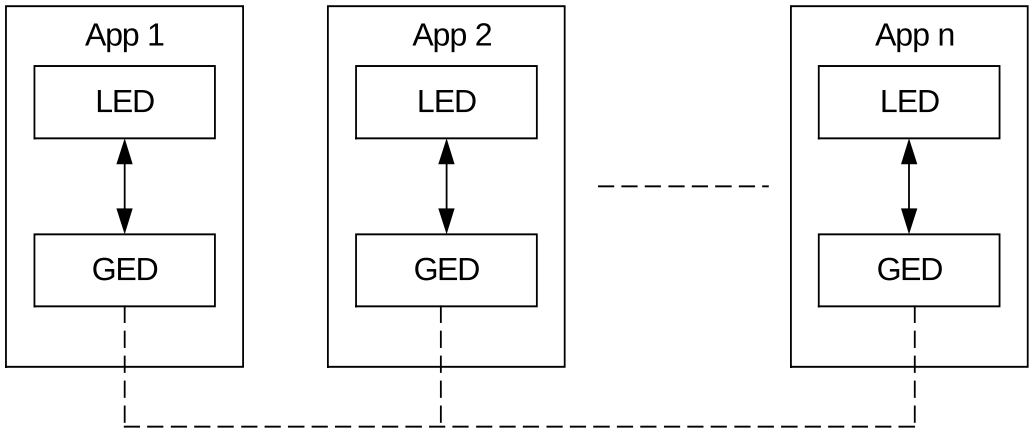


Figure 5.1. Architecture of a distributed event detector system.

In this approach, each application communicates directly with every other application that is part of a distributed application scenario. Each application needs to know other applications in order to communicate with them. As shown in figure 5.1, in this approach, both local and global events are detected in a local site, and applications are coupled with each other to exchange messages. [7] Each application requires both a local event detector and a global event detector module, and has to directly communicate with every other application as shown in figure 5.2. In addition, this approach requires that all applications to be running at all times. Most of the time, event detection in each site keeps on flooding the communication channel with messages rather than effectively monitoring the events of interest. In order to detect global events, each consumer application directly sends requests to appropriate producer applications in which its constituent events are defined. When an event occurs, a producer application sends a notification message back to a consumer application. Each application is responsible to detect a composite event by itself.

The following example (shown in figure 5.2) is used to demonstrate how this approach works and derive the communicate cost for detecting a composite event in order to compare with the other approaches. A local event LE1 and LE2 are defined as primitive events in application (called Prod1 and Prod2 respectively). The Cons1, Cons2, and Cons3 applications are interested in a composite event, which is composed by LE1 <op> LE2. Each application needs to send event detection requests to Prod1 and Prod2. When the event occurs, the producer application sends the notification message to notify the consumer applications. The total number of communication in this scenario is: 12 (6 detection request messages and 6 notification messages)

An event operator can be either binary or ternary. A number of consumer application (Cons1, Cons2, Cons3, …, ConsX) may be interested in the same event. Each consumer sends an event detection request to Y producers where Y is either 2 or 3 (binary or ternary operator).. When the event occurs, each producer sends back an event notification to all registered consumers. Therefore, the number of messages sent is: 2 * X * Y where the upper case X is used to represent a number of consumer applications that are interested in the same global composite events.

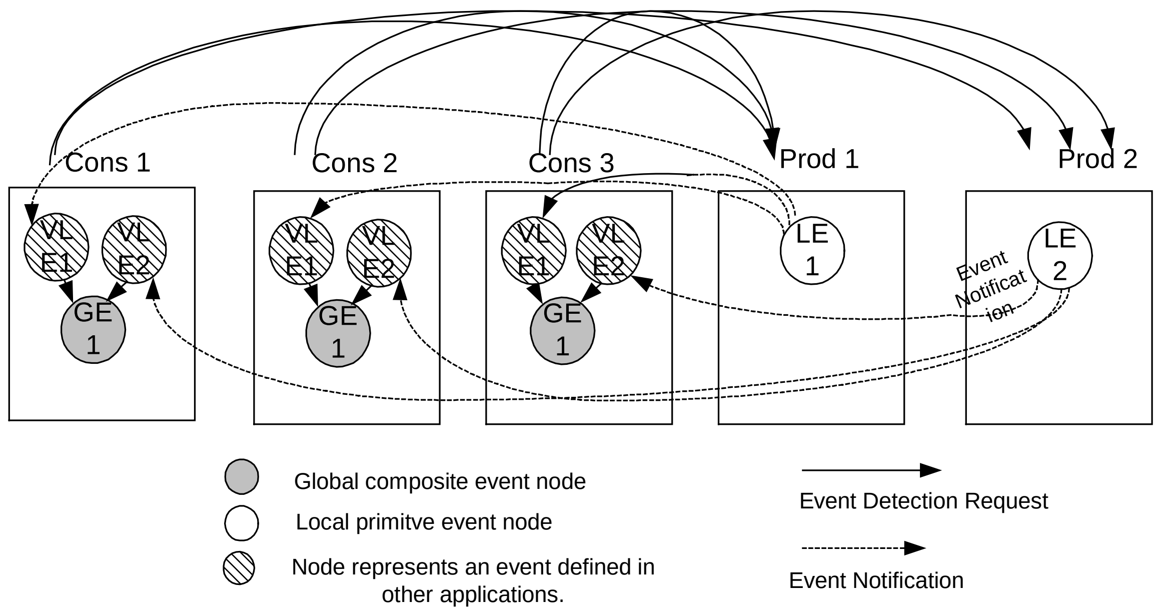


Figure 5.2. Global event detection in a distributed event detector.

5.3.2 Client/server architecture

This approach introduces a global event detector as a server. Each application only communicates with the server and hence does not have to know the identities of each application. The server is responsible for managing the subscription/notification aspect of global event detection. Determining what global events to detect and where to detect, the server allows clients (applications) to share information. The global event detector partially relieves the applications of the event detection. Each application still contains the local event detection module.

Applications, in this approach, are loosely coupled. A consumer (of events are detected in another application) application can subscribe to remote events through the global event detector. Running in the background on the server, the global event detector keeps track of subscriptions on the server site, and forwards the request to the appropriate application that generated the event (a producer). Once the event occurs in the producer site, the global event detector is notified which forwards the notification to the consumer.

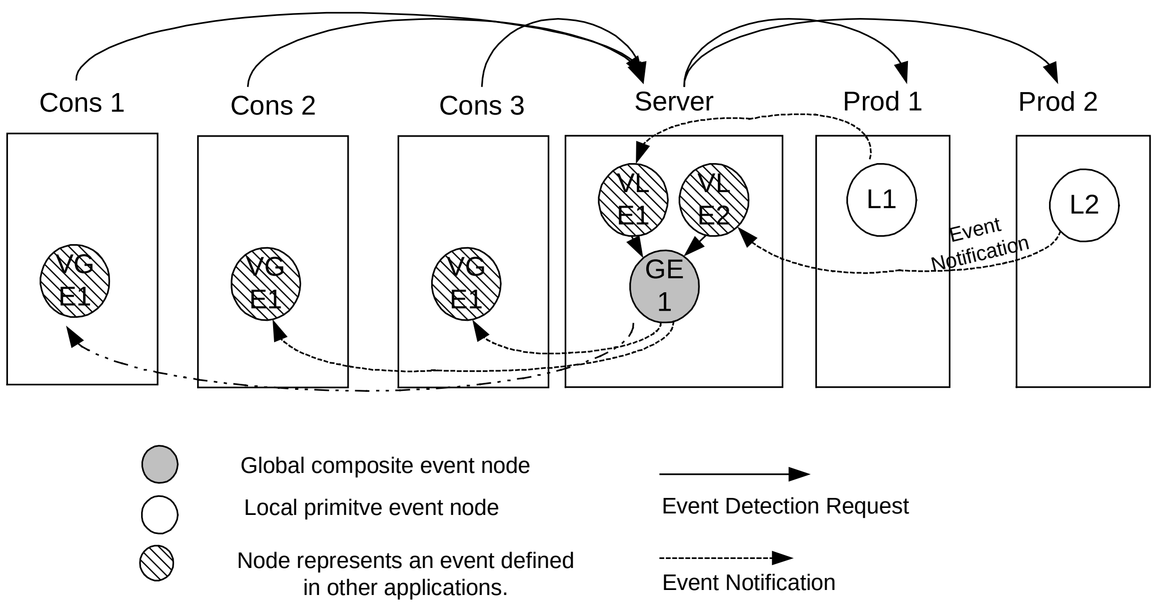


Figure 5.3. Global event detection in the global event detector.

For the earlier example, using this approach, the number of messages among applications and the server is: 10 (5 notification request messages and 5 notification messages). In general, the number of messages between clients and server is: 2*[X+Y] where X is used to represent a number of consumer applications and Y is either 2 or 3 (binary or ternary operator). The number of messages in this approach will be less than that of the first approach when the number of consumers increases.

The first approach works well only with very small number of applications and small number of global events. The client/server infrastructure decreases network traffic as compared to the first architecture, especially, when the applications and number of global events are more.

The communication between clients and the global event detection server plays an important role in system performance. There are three communication approaches discussed in the following section:

5.3.2.1 Client/server with remote procedure call architecture

Remote Procedure Call (RPC) is a client/server infrastructure that allows clients to invoke a method residing in the server in a different address space (either on the same machine or a remote machine). It encapsulates the details of the network interfaces and operating system from the applications. Most RPCs follow the blocking or synchronous protocol. The client application is blocked after sending the request to the server, until a return is received.

Two way remote procedure call (RPC) approach works only in situations that allow only one application per machine. The server cannot differentiate multiple clients/applications running on the same machine since they all obtain the same client identification (host name, service program number, version number, procedure number). Even though the remote procedure call (RPC) along with a socket-based approach works properly in the global event detection in C++ version, it is not suitable for application using the object-oriented paradigm. RPC doesn’t support object communication between application residing in different address space. Therefore, it is complicated to extend the earlier version of Global Event Detector to support an object-oriented application.

5.3.2.2 Client/server with middle-oriented middleware architecture

Middle-oriented middleware (MOM) [15] is an alternative infrastructure, which operates as a mediator between the server and client. MOM, and resides on both sides of client and server, is responsible for transferring messages between them. Most MOM architectures support both synchronous and asynchronous communication. Asynchronous communication allows clients and server to send messages in non-blocking manner

In this approach, MOM provides message router and message queues for asynchronous message passing between applications and the GED server. These administrated objects must be configured before running the global event detector. After starting MOM message router, applications can send an event detection request message to GED through the channel or queue. Similarly, the GED server can send the notification message back to applications via an asynchronous call. This kind of communication enables the server to continue performing other tasks without waiting for response from applications. The scenario of this approach is shown in figure 5.4

The benefits of this architectural approach are as follows. Clients are enabled to exchange the data with server in synchronous or asynchronous manner. For asynchronous communication, once the client sends out a message through the message queue provided by MOM, it is allowed to handle other jobs without awaiting a response from the server. MOM can ensure guaranteed message delivery.

This approach seems to make system run effectively and robustly. However, there are many drawbacks in this approach. Typically MOM architectures are proprietary software from many leading vendors. Therefore, we cannot modify internal infrastructure to accomplish our goals. Building the GED on the top of MOM incurs substantial overhead since it already provides rich set of functionality, such as, transaction management and security control, which are excessive for the GED server. In addition, duplicate subscription information is stored in both MOM and GED. Since substantial resources are shared by MOM, it could deteriorate system performance. Hence, this approach is not suitable for our development.

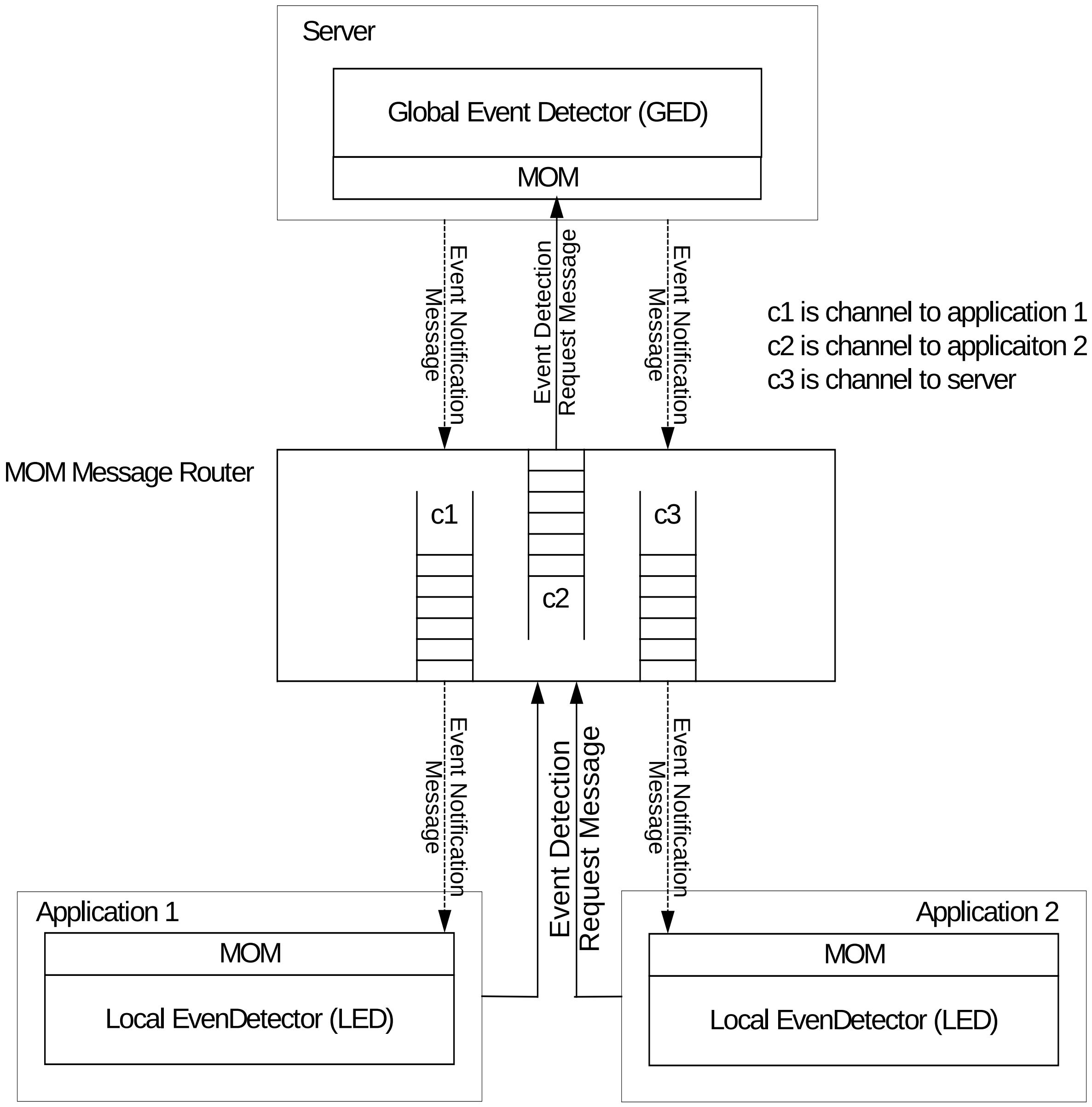

Server
Global Event Detector (GED)
MOM
Event Notification Message
Event Detection Request Message
Event Notification Message
c1 is channel to application 1
c2 is channel to applicaiton 2
c3 is channel to server
MOM Message Router
c1
c2
c3
Event Notification Message
Event Detection Request Message
Event Notification Message
Application 1
MOM
Local EvenDetector (LED)
Application 2
MOM
Local EvenDetector (LED)

Figure 5.4. A design model of GED with message-oriented middleware.

5.3.2.3 Client/server with object request broker architecture

An object request broker (ORB) is also a middleware that manages interaction between clients and servers, preserving full object messaging semantics. The objective of ORB is to support object communication across applications on different machines. Clients can request services on the server through interfaces, which are provided by the server. Working in conjunction with ORB, client applications can invoke methods that are defined in a remote application as if those methods are defined in their own applications. In addition, clients are not concerned about the location and implementation of services. Implementation of services can be modified to improve performance or revised with a better algorithm without affecting client applications. This transparency enhances system modularity and maintainability.

Three of major ORB technologies are Microsoft's Distributed Component Object Model (DCOM), [16] The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA). [2] and JavaSoft's Java/Remote Method Invocation (JAVA/RMI).[17] DCOM doesn't support multiple inheritances. It provides various functionalities by using multiple interfaces. It is broadly used in PC platforms. Unlike DCOM, both CORBA and Java/RMI support multiple inheritances. In addition, both of them are platform independent. However, CORBA doesn't support distributed garbage collection, since not all the languages support garbage collection, In addition, the license costs and compatibility of the vendor-dependent CORBA products are limited to our development. In contrast, JAVA/RMI are freely available. Adopting JAVA/RMI, we can run the application on any platform as long as the Java Virtual Machine (JVM) is installed.

JAVA/RMI provides mechanisms for manipulating remote distributed objects, and retrieving the class files from the Java/RMI server dynamically. Additionally, JAVA/RMI can be easily integrated with the existing system, a local event detector. Consequently, this approach is the most appropriate to implement the global event detector.

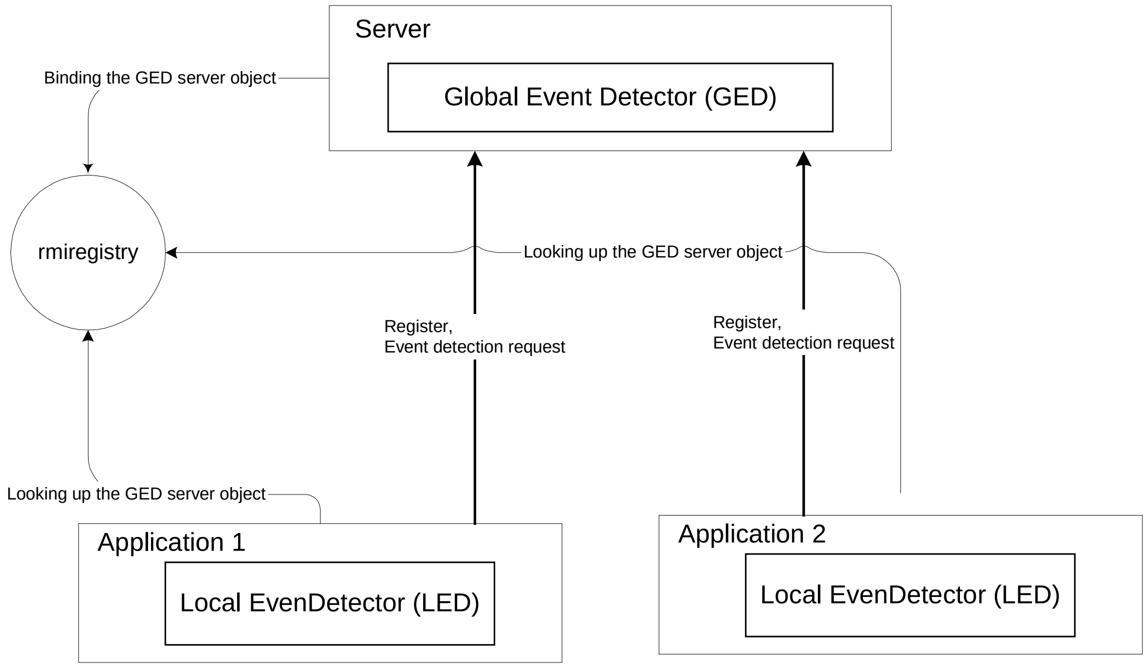


Figure 5.5. A RMI communication design model of GED from applications to the GED server.

To implement the distributed object applications, we need to be able to locate the remote objects, communicate with remote objects, and load class byte codes for object communications. The rmiregistry in RMI provides a naming service and allows the clients and servers to rebind and lookup the remote objects. Concepts of stubs and skeletons are used to handle synchronous communication between applications. When the client invokes the remote method, the stub is responsible for initiating a connection, marshaling the parameters to the remote virtual machine, and unmarshaling the return value. On the other side of communication, the skeleton unmarshals the incoming parameters dispatches the request to the actual implementation, and unmarshals the return value before sending it back to the caller.

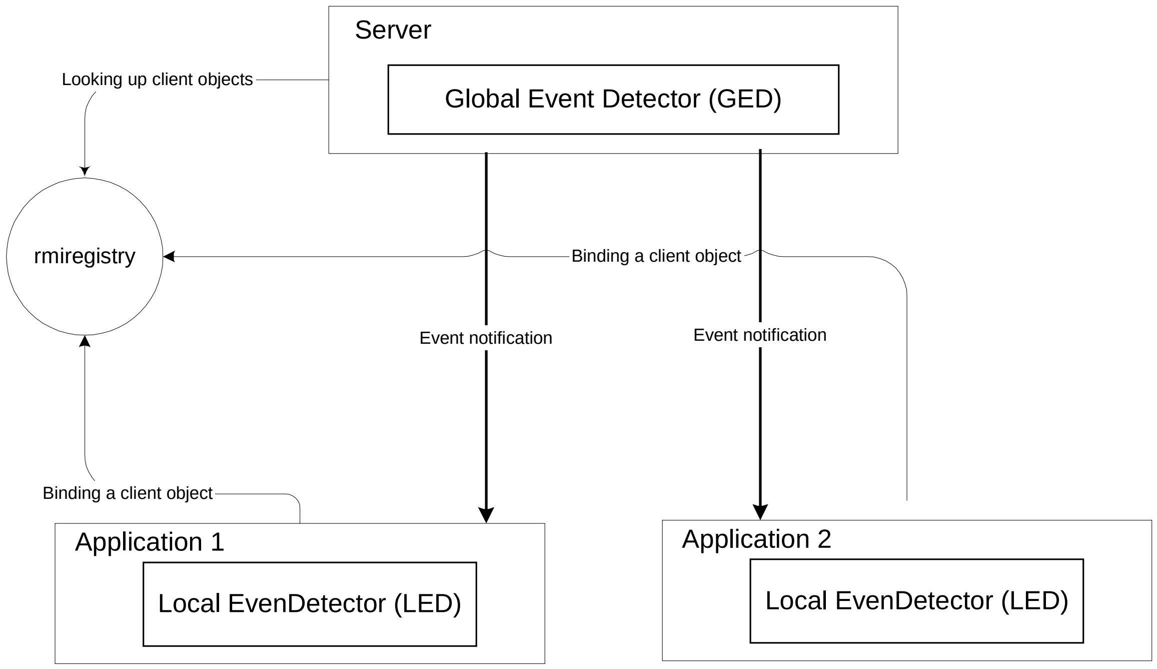


Figure 5.6. A RMI communication design model of GED from the GED server to applications.

Client/server with two-way RMI communication approach is used to implement global event detector. Figure 5.2 illustrates how clients can register to the GED server through RMI communication. When the GED server is started, it binds the GED server object to the rmiregistry. To obtain the GED server object, clients can look up from rmiregistry. After obtaining the reference, clients are entitled to register to the GED server and subscribe to the events of interest. On the other hand, figure 5.6 demonstrates how the GED server can notify the clients when the events occur. To receive the notification, clients need to bind the client objects to rmiregistry. Once the event of interest is raised, the GED server looks up the client object and notifies the client.

5.4 Java Remote Method Invocation

The Java Remote Method Invocation (RMI) is used in the global event detector to send detection request and notification messages. Clients also use the remote method invocation to register with the GED server. The RMI is a backbone for the communication mechanism of GED system. This section summarizes the overview of the RMI.

According to the Java specification, a *remote object* is one whose methods can be invoked from another Java virtual machine, potentially on a different host. [java specification]. Clients can acquire this remote object to request for services. These services are obtained through methods of the remote object declared in one or more remote interface. RMI uses *stubs* and *skeletons* to communicate with remote object. The stub, residing on the client, works as a representative for the remote object. When the client invokes a remote method, it actually invokes a method on the local stub, which is responsible for hiding the network layer, forwarding the call to the remote object, and returning the result to the client invocation. The skeleton is used to dispatch the call to the actual remote object implementation on the server site, and send the result back to the client.

Based on Java remote method protocol, RMI allows objects to be marshalled and unmarshalled through object serialization technique. The key of object serialization is to store and retrieve objects in a serialized form. This self-describing byte stream can be used to sequentially transmit an object between two java virtual machines, and reconstruct the object from the stream.

5.4.1.1 Basic RMI programming

The following section describes how to develop the distributed application via RMI:

The first step is to define the scope of the services in remote interfaces. Each service corresponds to a method declared in the interface. Clients can request for services by invoking these methods. Here is a skeleton of interface definition for the remote interface. The interface contains methods which are invoked remotely.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface RemoteObjIntf extends Remote {
        Object service1(Parameter p1, Parameter p2, …) throws RemoteException;
        Object service2(Parameter p1, Parameter p2, …) throws RemoteException;
}
```

The next step is to implement a class of a remote interface. It needs to provide an implementation for each remote method defined in the interface. The example of implementation follows:

```
import java.rmi.*;
import java.rmi.server.*;
import compute.*;
public class RemoteObjImpl  extends UnicastRemoteObject
                            implements RemoteObjIntf {
        public RemoteObjImpl () throws RemoteException {
                super();
        }
        public Object service1(Parameter p1, Parameter p2, …) {
                …
                …
        }
        public Object service2(Parameter p1, Parameter p2, …) {
                …
        }
    }
```

It is necessary to install a security manager and register at least one remote object into the *rmiregistry*. To register, we have to bind the server object with a naming service. These can be done in the server main program so that the remote clients can later perform a look up.

```
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "//host/remObjName";
        try {
            RemoteObjImpl remObj = new RemoteObjImpl();
            Naming.rebind(name, remObj);
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
```

In the client site, the caller can look up for the remote object through *rmiregistry*. After getting the remote object reference, the caller can invoke a remote method as it is implemented locally.

```
    import java.rmi.*;
    import  RemoteObjIntf.*;
```

```
public class ComputePi {
public static void main(String args[]) {
        try {
                String name = "//hostname/ remObjName ";
                RemoteObj remObj = (RemoteObj ) Naming.lookup(name);
                remObj.service1();
                remObj.service2();
        }
        catch (Exception e) {
                e.printStackTrace();
        }
}
```

5.5 Global Event Detection Site

In a distributed setting, data is exchanged among applications. The communication strategy is a key factor to reduce the communication cost, and it is critical to system performance. The cost of communication can be described as:

Communication Cost = Frequency * (Overhead + Occupancy)

The overhead is the time to initiate the transfer. The occupancy is defined as the time it takes for transmitting the data. And the frequency is the number of times a message is sent. [18] These factors influence system performance. The system should minimize the size and number of messages exchanged among applications to obtain the better performance.

In the global event detector, there are two approaches for detecting global composite events. [7] The first approach is that all the global composite events are detected at the local site. The GED server acts as mediator to receive the information about the occurrences of constituent primitive events and forwarding the occurrence to all its subscribers. In this case, the event graph of the global composite event is constructed and events detected at the local site. In the second approach, the global composite events are detected server sites. The GED server not only receives and forwards the information, but also detects the composite event in some cases. Unlike the first approach, the GED does not forward every occurrence of the constituent primitive event to its corresponding application. If a part of the event graph is constructed on the server site, the GED sends the notification back to the corresponding client application only when the composite global event is detected. Therefore, the location where the global composite events are detected has a major impact on the number of messages passed between the server and client applications. Consequently, the second approach is implemented in the global event detector in order to decrease the communication cost. The criteria where the event graph should be defined and detected are described in the following section.

A global composite event can be detected either at the local site or the server site. The site where a global composite event is detected is determined by its constituent events. The global composite event is detected at the GED when all of its constituent events are the global events; whereas, it is detected at the local site when one of its constituent events is a local event. The following examples show how and why this is made.

5.5.1 Global composite detection at the GED server

This section compares the communication cost of detecting a global composite event at the server versus its detection at the local site. We conclude that the global composite event should be detected at the server when all the constituent events of the composite event are the global events.

In the example, an event *E1* is defined as a primitive event in one application called *Prod1*, and an event *E2* is specified in another application called *Prod2*. Running in different address space, the *Con1*and *Con2* applications are interested in a composite event, which is composed by *E1* <op> *E2*. There are two places where the composite event can be detected.

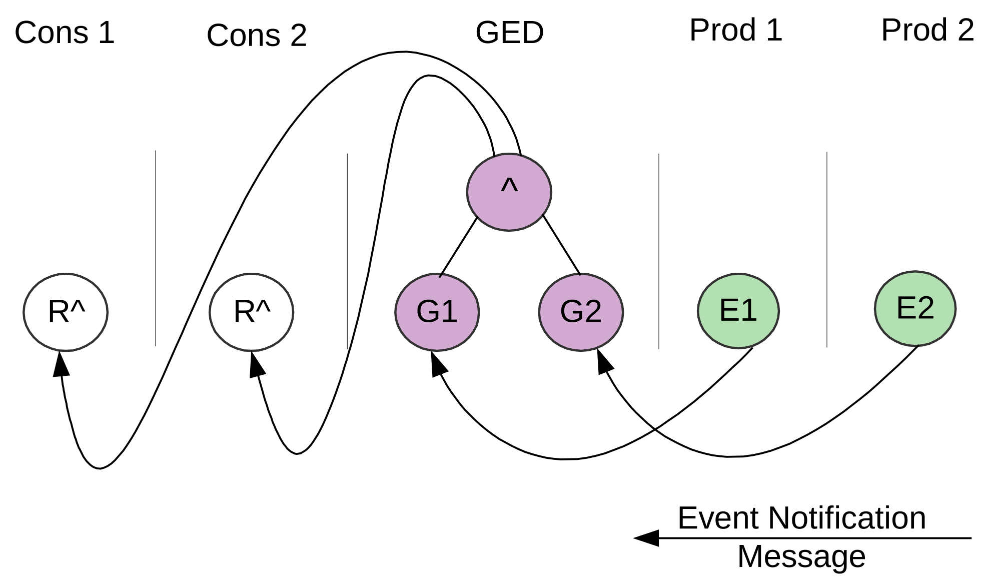


Figure 5.7. Composite event detection at the GED server when all the constituents are global events.

- **Global composite event is detected at the GED**

Figure 5.7 illustrates event graphs of each application and the GED server. In this case, the composite event graph is constructed and detected on the server site. These two events (*E1* and *E2*) are considered to be global primitive event from the viewpoint of consumer applications. Both consumer applications send the event detection request message to the GED server. Then the GED server constructs event graph and forwards the request to the corresponding producer applications. In the server site, *G* node is used to represent the event on the producer site. And *R* node on the consumer site represents the remote event. Once *E1* occurs, the GED server is notified from the prod1. Prod2 also notifies it when E2 occurs. Then the server propagates the occurrences of the constituent events into the internal node of the event graph before sending the occurrence of *AND* (^) event to each consumer. Consequently, the communication cost for detecting this composite event is equal to four message transmissions.

To consider in general case, E is a constituent event of a composite global event and is defined in a producer application. The constituent events are defined in *Y* producer applications. When event occurs, *Y* producers send notification messages to the GED server. After the GED server detect a composite event, it will signal an occurrence of event to all subscribed consumers. The communication cost for detecting a composite event is *X*+*Y* where *X* is the number of consumers for that event and *Y* is either 2 or 3 (binary or a ternary operator).

- **Global composite event is detected at the local site**

Figure 5.8 illustrates the same scenario as above, but the global composite event is detected at the local site. In this case, the event graph of the composite event is constructed on the consumer site. When E1 and E2 occur, the GED server simply forwards the occurrence of the events E1 and E2 to the corresponding consumer. Then the consumer application propagates the occurrence of the primitive events into the internal node and detects the global composite event in its local site. The communication cost for this case is equal to six message transmissions.

To consider in general case, the communication cost for detecting a composite event is : Y + (X*Y) where x, y, are the same as before.

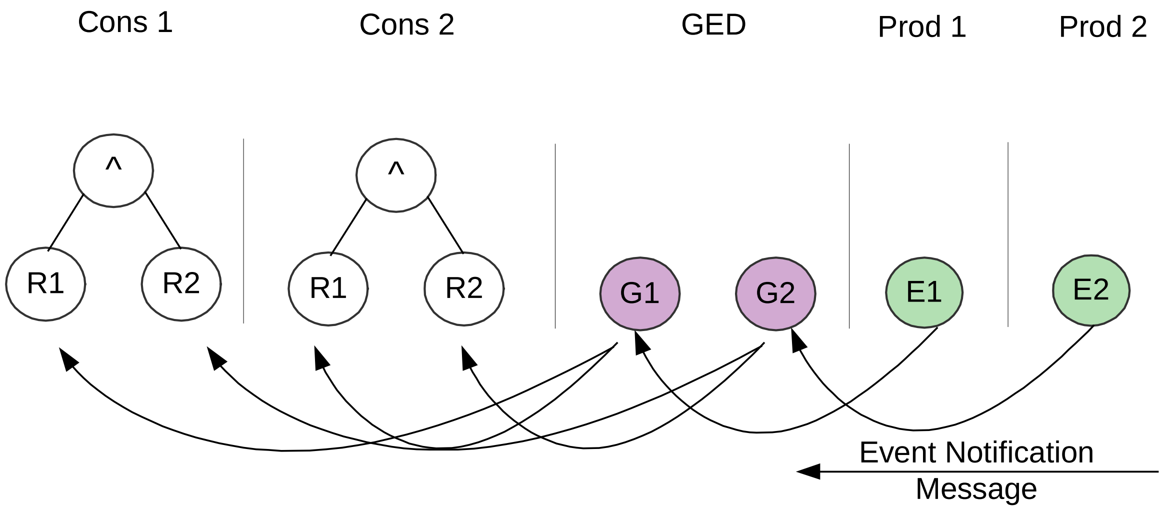


Figure 5.8. Composite event detection at the GED server when some of the constituents are local events.

The 2 expressions are same when x is 1 and y is 1. Y is never one as it is an operator. So, even for 1 consumer (when X is 1), one can see that it is beneficial to detect the event at the GED.

As a result, the number of messages between clients and server of the first case is less in the first case than in second case. Therefore, the global composite event should be detected at the GED when all of its constituent events are global events.

5.5.2 Global composite detection at the local application

This section compares the communication cost comparison to determine when a global composite event should be detected at the local application. It turns out that when at least one of the constituents is a local event (local primitive or local composite event), it is beneficial to detect the event at the local site.

In the example below, the consumer (*Cons1*) is interested in the global composite event composed of a local event, and a remote event *E1* that is defined in the producer (*Prod1*) application. A *L* node on the consumer site denotes the local event.

- **Global composite event is detected at the GED**

Figure 5.9 illustrates the composite event detection at the GED. In this case, the producer first defines the primitive event (E1) in Prod1. Next, the consumer defines the local event (L1) and specifies the composite event on the events L1 and E1. The composite event graph is constructed on the server side. When E1 occurs, the producer sends the notification message to the GED. After receiving the notification message about L1, the GED detects the composite event and sends the notification message back to the consumer. Consequently, the communication cost for detecting this composite event is equal to three.

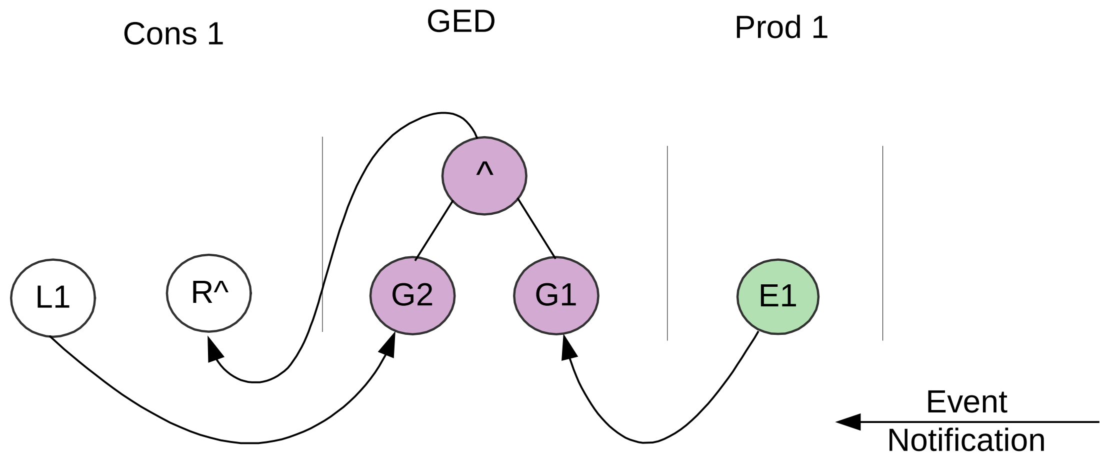


Figure 5.9. Composite event detection at the GED server when all the constituents are global events.

To consider in general case, En is an event which is defined in the nth producer application (termed prod(n)).We can n to represent a number of constituent events that defined outside a consumer application. Whenever event occurs in either consumer or producer applications, the occurrence of event has to be sent to the GED server. Therefore, n notification messages from producer applications are sent to the GED server, and Y- n notification messages are sent from the consumer application where Y is either 2 or 3 (binary or a ternary operator). After the GED server detects a composite event, it sends a notifications message back to the appropriate consumer. The communication cost for detecting a composite event is (Y-n) +n +1 or Y+1.

- **Global composite event I s detected at the local site**

Figure 5.10 illustrates the composite event detection at the local site when some of the constituent events are local events. In this case, similar to a previous one, the producer first defines the primitive event (*E1*); the consumer defines the local event (*L1*) and specifies the composite event on the events *L1* and *E1*. But the composite event graph is constructed on the local site. Therefore, the consumer application does not need to send the event notification message of *L1* to the GED server. When *E1* occurs, the producer sends the notification message to the GED. Then the GED forwards the occurrence of *E1* to *Cons1*. Consequently, the communication cost for detecting this composite event is equal to two.

To consider in general case, En is an event which is defined in the nth producer application (termed prod(n)).We can n to represent a number of constituent events that defined outside a consumer application. When event occurs, a producer application sends a notification message to the GED server, and it will forward the message to a consumer. The communication cost for detecting a composite event is 2(Y–n) where Y is either 2 or 3 (binary or a ternary operator).

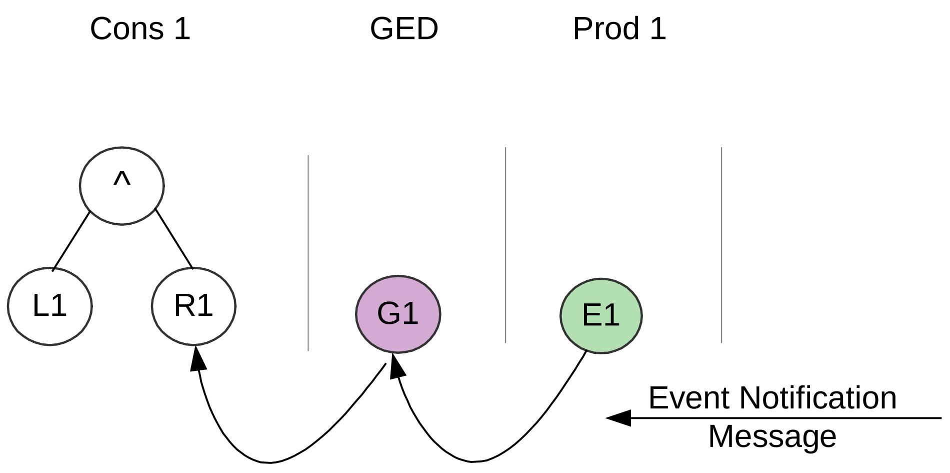


Figure 5.10. Composite event detection at the local site when some of the constituents are local events.

The 2 expressions are same when Y is 3 and n is 1. Beside that, the communication cost of detecting in a local site is less than that of detecting at the GED when at least of a constituent event is defined and detected inside a local application..

As a result,the communication cost for detecting the global composite event at local site is lower when some of constituent events are local events. Therefore, the global composite event should be detected at the local site when some of constituent events are local events.

5.6 Extension of Local Event Detection

This section describes how the local event detector can be extended to accommodate the global event detection. In addition, the key classes are also explained..

5.6.1 Type of events

An event is an occurrence of interest at a specific point in time. With in an address space, an event is simply classified into primitive and composite events. In Java, method invocations are considered as primitive events. There is no notion of local and remote event. All the method invocations that are monitored reside in the same address space as the application. In a distributed environment, one application may be interested in the event defined in another application running in another address space. This kind of event is called global event as described in the chapter on semantics of events. Figure 5.11 depicts the event class hierarchy. The next section describes event classes that are designed for representing four different types of event (local primitive, local composite, global primitive and global composite).

5.6.1.1 Event

Event class, an abstract class, contains common attributes and behaviors, including abstract methods, which are used for representing the event node in the event graph. In a distributed environment, an event defined in one address space can be subscribed to by any application. Therefore, the event detector should be able to determine whether an event (or a corresponding node in the event graph) is subscribed to by another application. The *forwardFlag* is added into Event class for this purpose. In the GED server, once it receives a notification message from the producer application, it decides to send the notification message to consumer applications by using *sendBackFlag*. This flag is set only when there is at least one rule defined in the event node. This Event class has Primitive, Composite, Remote, and Global event class as subclasses.

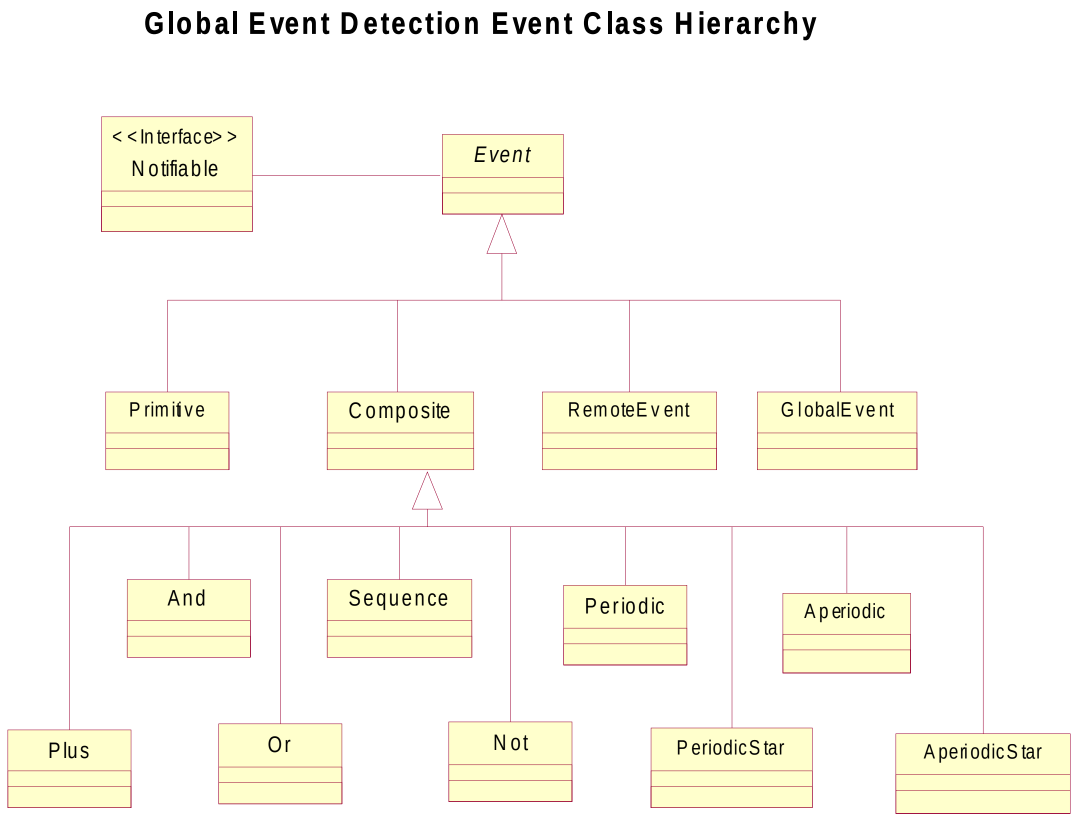


Figure 5.11. Global Event Class Hierarchy.

5.6.1.2 Primitive

Primitive class was introduced in the LED. Each *Primitive* object is used to represents a local primitive event in the global event detection.

5.6.1.3 Composite

Composite class is also an abstract class. The composite events are defined by applying one event operator to primitive events and/or other composite events. The composite class is subclassed to each event operator (AND, OR, SEQUENCE, NOT, PLUS, APERIODIC, APERIODIC*, PERIODIC, PERIODIC*). Composite events are classified into two groups (local composite and global composite event). If one of its constituent events is a global event, it is a global composite event. Otherwise, it is a local composite event. However, for the maintenance purpose, each operator object is used to represent both a local composite and a global composite event instead of having one for the local composite and one for the global composite event. The site of the global composite event detection is determined by event specification at run time. Since our design has only one class for both the composite event node constructed by LED and the one constructed by GED, the *EventCategory* class is used to categorize the event type. We use this to customize the functionality for each type in the program. For example, when an event is detected and the event category of that node is equal to *EventCategory.GLOBAL*, the *executedRules( )* method should be ignored. This is because there is no rule defined on the *global* event (at least for the present) at the GED server.

5.6.1.4 Global

As defined earlier, a global primitive event is an event that is defined and detected outside the current application. The GED server not only forwards the messages between applications, but it is also responsible for detecting global events Therefore, the global event class is introduced to represent the global primitive event node on the server site. Global event nodes are used to construct a global event graph for detecting composite events.

In order to subscribe to an event defined outside a current or local application, it current or local application has to know the *producer event name*, *remote application name*, and *host name*. The *producer event name* is the event name that is defined in the other application where the event is detected. The *application name* and *host name* denote the application and machine name where the event is defined. This information is necessary for the mediator to forward the detection request to the producer application and send back the notification message to the corresponding consumer application.

5.6.1.5 Remote

A remote class has been added into the event class hierarchy. Each Remote object corresponds to a global event (global primitive or global composite), which is defined in another application. It is similar to the Global object, but is used to represent the global event in the consumer application. The Remote object contains the information about the global event. When representing a global primitive event, producer event name, remote application name, and host name of the producer application are stored in the Remote object. The application name and host name are ignored, when the Remote object corresponds to the global composite event.

It would be tedious to refer the primitive global event by using these three attributes whenever the consumer application tries to refer to the Remote object. Remote object introduces a consumer event name (termed *consName*) to store a proper and meaningful name that is internally used inside the consumer application instead of using concatenation of producer event name, remote application name, and host name.

5.6.2 Communication Module

The existing local event detector is only aware of the events defined in its address space. It cannot send the event detection request to another application. Therefore, a communication interface has been introduced to handle the remote call so that the existing local event detector can exchange information with other applications. Figure 5.13 illustrates the overview of how one application can interact with another application.

The LED interface is introduced to facilitate the communication between LED and GED. It is responsible for looking up for the remote objects as mentioned in the RMI section, and making remote invocations to send an event detection request or event notification to the server. It also has a listener that waits for incoming messages such as event notification and event detection request from the server. The class diagram of *LEDInterface* is shown in figure 5.12.

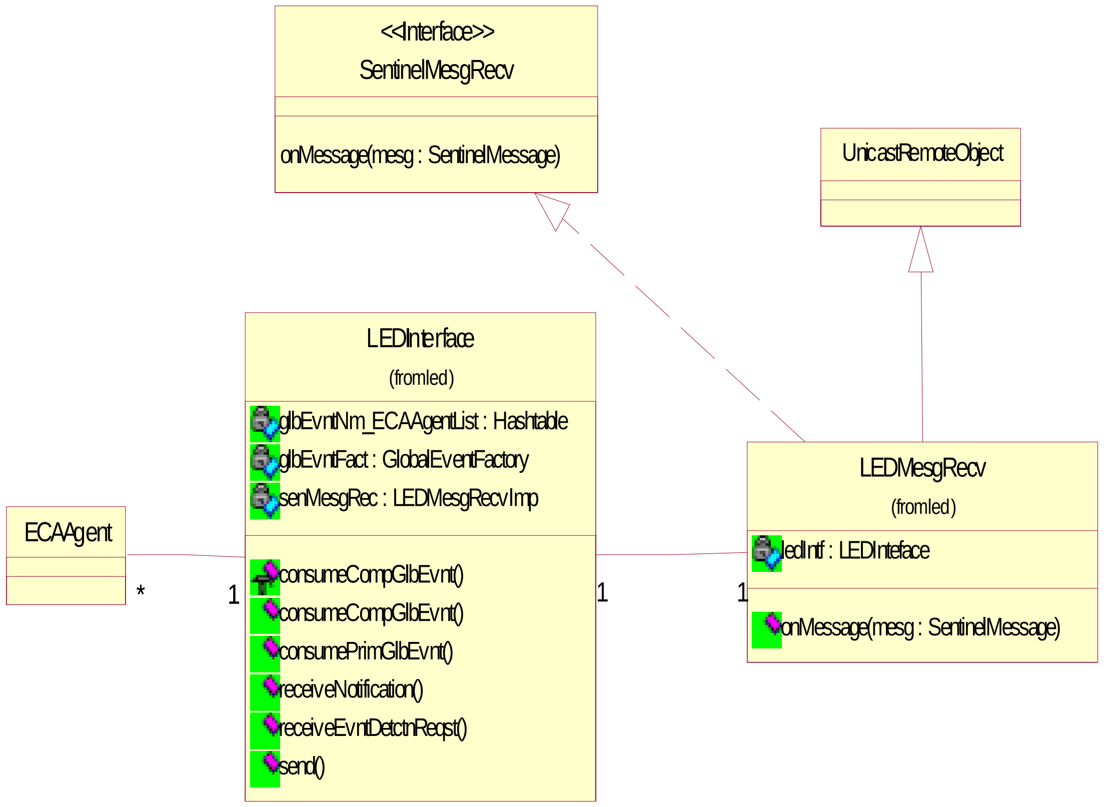


Figure 5.12. The LED interface class diagram.

On the server site, the GED interface is designed not only to forward the messages from one application to another application, but also to store some data to construct the global event graph and additional information for global event detection. Every application needs to register with the server through this interface. The communication layer between GED and LED is shown in figure 5.13.

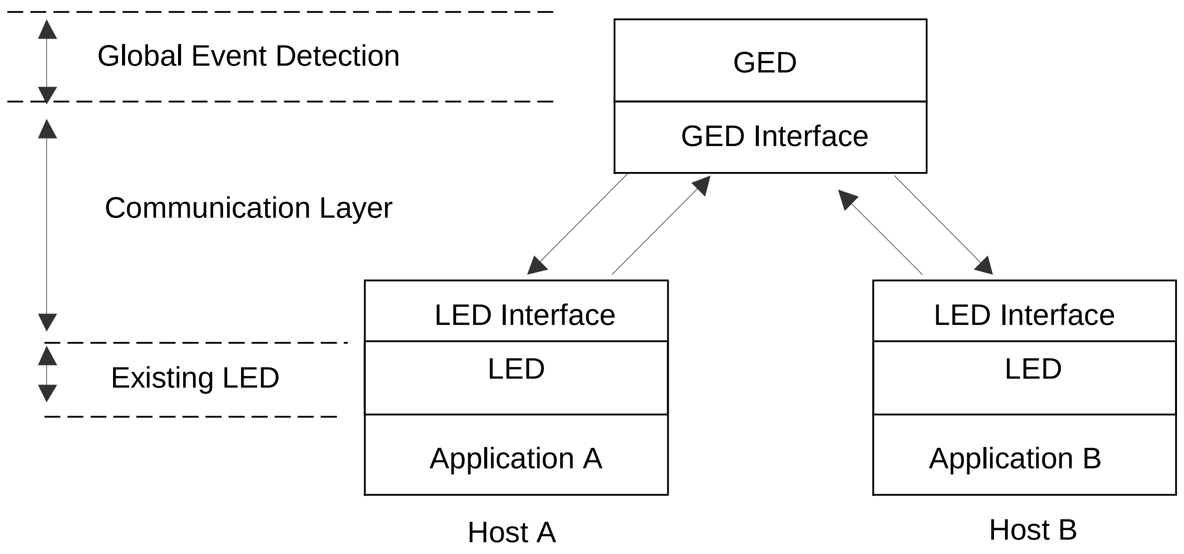


Figure 5.13. The communication layer between LED and GED.

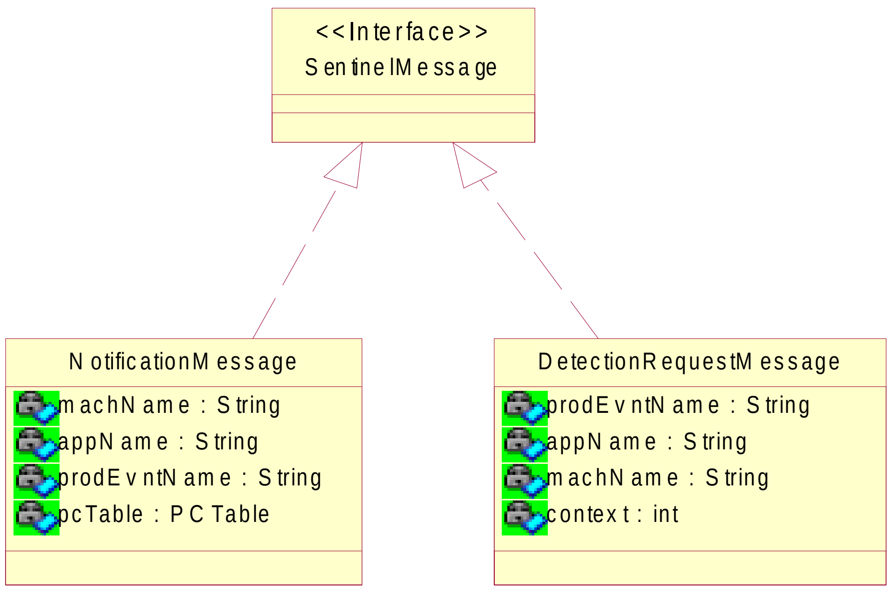


Figure 5.14. Sentinel Message class diagram.

5.6.3 Type Of Messages

Typically, a client application makes a remote invocation to the server to request the detection of a global event, and receive event notification from the server when that event occurs. As shown in figure 5.14, there are two types of messages (*eventNotificationMessage* and *detectionRequestMessage*) that are exchanged among clients and the GED server.

5.6.3.1 Event Notification Message

When a client send a request to the global event detector, it is necessary to send the event name, application name, and host name of the global event to the server. This request also includes context bit information to capture the useful semantics of the application. When a rule is defined on a global event in a particular context, the context data will be sent to the GED server. This information is packed into a message called *detectionRequestMessage*. Once the GED server receives the message, it creates the global event node representing the event on the server site. The *sendBackFlag* described earlier is set in order to specify that there is at least one application listening to the occurrence of this event. Then the server forwards the message to the application that defines the event. The producer application unpacks the package and sets the *forwardFlag* associated with event node to be true so that it will signal the event occurrence to the server.

5.6.3.2 Detection Request Message

Similar to *detectionRequestMessage*, the *eventNotificationMessage* contains information about the global event. However, it contains the *PCTable* information instead of the context bit. This *PCTable* contains the list of parameter lists. Typically, the relevant information is recorded in the parameter list when the primitive event occurs. The occurrence of this primitive event is propagated to the internal node if it is a constituent event of the composite event. In addition, the parameter lists are inserted into the event table of internal node. Whenever the *forwardFlag* of the notified node is set, the *evntNotificationMessage*, including the *PCTable*, will be sent to the GED server.

5.7 Global Event Graph

As described in the chapter 4, the event graph is used for detecting composite events. Each event that is defined in the application is represented as an event node in the graph. The relationship among composite events and their constituent events forms an event graph. In the event graph for LED, each node has a list of event subscribers and a list of rule subscribers. When a composite event subscribes to its constituent nodes, the reference of composite event will be stored into the list of event subscribers. Similarly, when a rule is defined, it is stored in the list of rule subscribers.

As mentioned earlier, the composite global events can be detected either on the local site or on the server. The composite global event can also be detected on the GED server by using the event graph as shown in figure 5.15. The leaf nodes of the global event graph represent the global primitive events. The internal nodes represent global composite events. Unlike the node of a local event graph, the global event node does not have a list of rule subscribers, since all the rules are locally executed at the application site. The rules that are defined on a global event are applied to the *Remote* event node on the application site instead. Each global event node contains only a list of event subscribers, which contains references to the global composite event nodes. In addition, each node maintains occurrences of events and their parameter lists in the *PCTable*. Whenever a global event occurs, it will check the *sendBackFlag* to check whether to send the event notification message toe appropriate clients (or consumers).

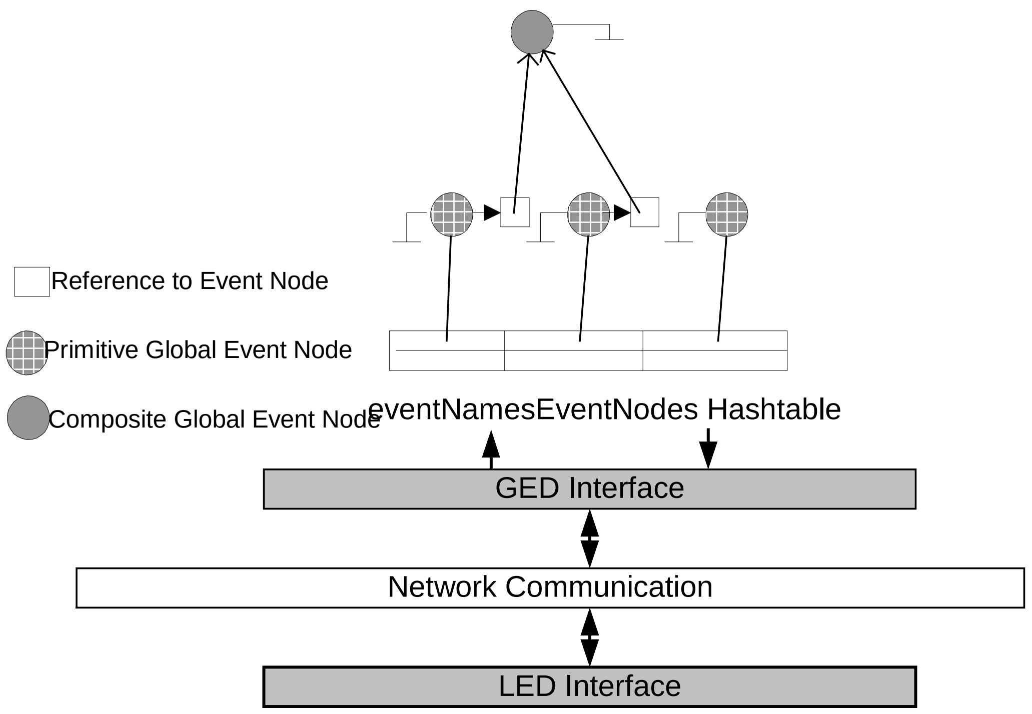

Reference to Event Node
Primitive Global Event Node
Composite Global Event Node
eventNamesEventNodes Hashtable
GED Interface
Network Communication
LED Interface

Figure 5.15. Global event graph for detecting global events.

5.8 Application Configuration File

As described earlier, the GED is extended from the LED to provide the active capability in the distributed environment. Both of them are integrated into the *sentinel* package from which applications import to obtain the services. However, users should be able to tailor the execution environment to meet the particular requirements without burden to system resources, since the overhead associated with the global event detection is higher than that of local event detection. For instance, the LED interface is instantiated to interface with the GED server in the global event detection, whereas the communication layer is concealed from the application in the local event detection. A stand-alone application should work in conjunction with the local event detector to avoid the overhead of the global event detector.

Since the global event detection involves applications from many sites, the event name defined in each application is not sufficient to uniquely identify a global event in a distributed setting. As discussed, the application ID (application name and host name) is needed to disambiguate a global event. Each application can obtain the host name at run-time, getting the application name from the users. By providing an application name, users are allowed to run many applications on the same machine. Users also need to provide the GED server information, including the IP address and the GED server name before running the application so that the application can register and send requests to the appropriate server. The server name is not hard-coded since it is possible to have multiple GED servers. We plan on supporting multiple GED servers for scalability and for replicating global event detection.

The application configuration file is introduced so that users provide necessary information and customize the event detector for their needs.

5.9 Global Configuration File

In order to define a primitive global event, the application name and host name of the producer application have to be specified at the time the event is defined in the consumer application. Therefore, these applications are dependent on the machine. If these applications are ported to run on other machines, the machine name and application name that are defined in the event definitions inside each application needs to be modified in order to subscribe to appropriate application. To overcome this problem, a name mapping is provided to the GED server to make the application virtually independent of the application name and machine name. The users only need to specify the old application ID and the new application ID in the configuration file instead of re-writing the application code. In addition to the mapping information, the GED server name can be supplied into the global configuration file.

CHAPTER 6

IMPLEMENTATION OF GLOBAL EVENT DETECTOR

In chapter 5, we discussed the design and architecture alternatives. This chapter describes the implementation details of the global event detector. First, the configuration of each application and a GED server is described. Second, it describes how the applications and GED are initialized. Third, the event and rule definitions are discussed in detail. At the end, it explains how the global primitive and composite events are detected in the distributed environment.

6.1 Implementation of an Application Configuration File

The event detector can be configured to support either a stand-alone application or a distributed application through an application configuration file (*App.config*). In addition, the users can set up the event detector to run along with the rule scheduler or without it. For a stand-alone application, the default configuration will be set by the *sentinel* package if users do not provide the *App.config* file. Since the default configuration file is set to support only local event detection, the application that imports the *sentinel* package to obtain the active capability in the distributed environment needs to provide the *App.config* file. Furthermore, each application running as part of the distributed application needs to specify its application name and the location of the GED server. Two applications cannot have the same name within a machine, since the GED server cannot distinguish between them.

When an *ECAAgent* is initialized, it first reads *App.config* file from the current directory. The *readAppConfig( )* method in *Utilities* class is used for reading the properties of the event detector through *App.config* file, which contains the scope of event detection, the GED server information, and the local application information. All of this static information is stored in the attributes of the *Constant* class of the *LED* package. The event detection system will use this information at run time. The format of an *App.config* file is as follows:

SCOPE **[LOCAL/GLOBAL]**

RULE_SCHEDULER **[ON/OFF]**

GED_URL **[IP ADDRESS]**

GED_NAME **[GED NAME]**

APP_NAME **[APPLICATION NAME]**

Each of the parameters is described below.

- SCOPE

The scope of the event detector can be set to local or global, depending on the usage of application. For the stand-alone application, it should be set to LOCAL in order to avoid the overhead caused by the communication interface and the remote event manager.

- RULE_SCHEDULER

The rule scheduler is responsible for scheduling each rule and its execution according to the specified priority and coupling mode. The rule scheduler can be turned *ON* or *OFF*. When the rule scheduler flag is turned *ON*, the local event detector can trigger the rule according to its priority and coupling mode. Otherwise, rules are executed in the order in which they were defined for each event. Rules on primitive events are executed prior to rules on composite events.

- GED_URL

In order to communicate with the GED server, the application must specify the location of the GED server. The GED_URL, in IP address format, is used to specify the location of the server.

- GED_NAME

The GED_NAME is a name of the server object that is registered in the rmiregistry. This GED_NAME needs to be matched with the GED_NAME that is specified in the Global.config (will be discussed in the next section). This might be useful for the future development when there are many global event detectors. This makes the application scalable and provides replication and better performance for event detection.

- APP_NAME

The application name is used to identify the application. Setting the application name allows users to run multiple applications on the same machine. Users must give a name for each application.

The example of the *App.config* file for a distributed application is shown below. The application name is "*producer*". This application registers to the GED server running on "*129.107.12.243*" machine. The GED server name is "*ged1*".

```
SCOPE GLOBAL
RULE_SCHEDULER ON
GED_URL 129.107.12.243
GED_NAME ged1
APP_NAME producer
```

6.2 Implementation of the Global Configuration File

The global configuration file (*Global.config*) is used by the server to setup the global event detector and solve the naming dependency problem. When the GED server starts, it reads this configuration file from the current directory. This file contains the GED server name (*GED_NAME*) and mapping from an old application ID to a new application ID.

The server binds this *GED_NAME* to the GED remote object, and registers the remote object in the *rmiregistry*. The applications on remote hosts can look up the GED remote object by using *GED_NAME*, and send the detection request or notification messages. The *GED_NAME* value is stored in the *Constant* class in the GED package.

In addition to the server name information, mapping information of the old application ID and new application ID is read from this file. This information is stored in two hash tables (*OLDAPPID_NEWAPPIDMAP* and *NEWAPPID_OLDAPPIDMAP*), which are shown in figure 6.1. The purpose of using two hash tables is to speed up the mapping process from the old application ID to the new application ID, and vice versa. This information is crucial for the GED server in resolving the machine name dependency. The main reason for the mapping is to make the changes in the application ID transparent for porting. The mapping allows the applications to be executed on different machines without changing and re-compiling the application code. The GED server can act as if all applications still run at the same location. Hence, the global event nodes are constructed based on the old event specification, which is coded as part of the application. In addition, it keeps clients' application ID under the old application ID in every place.

OLDAPPID_NEWAPPIDMAP

Old Application ID	New Application ID
producer_newdelhi	producer_bangkok
consumer_newdelhi	consumer_tokyo
[appName_machineName]	[appName_machineName]

NEWAPPID_OLDAPPIDMAP

New Application ID	Old Application ID
producer_bangkok	producer_newdelhi
consumer_tokyo	consumer_newdelhi
[appName_machineName]	[appName_machineName]

Figure 6.1. Mapping information of the old application ID and new application ID and vice versa.

The *GEDInterface* play a role in mapping the old application ID to the new application ID and vice versa. Mapping is needed in several places. For example, whenever the old application registers to the GED server with the new application ID, the *GEDInterface* gets the old application ID of this client form the *NEWAPPID_OLDAPPIDMAP* table and registers this client in the old application ID. In addition, when an event notification message is sent from a producer application running on the new machine, this message is seemingly to be from the new application, but the *GEDInterface* maps this new application ID of the producer with the old application ID, and uses the old application ID to notify the global event node on the server. In some cases, the *GEDInterface* needs to substitute the old application ID with the new application

ID. For example, when the GED server sends the event detection request form the GED to the producer application, and the producer application runs on the different machine, the *GEDInterface* can look up for the new application ID from the *OLDAPPID_NEWAPPIDMAP* table and use this new application ID to locate the remote object before making RMI call to subscribe to the event. The format of Global.config file is below.

```
BEGIN
MAPPING  [OLDAPPLICATIONID] [NEWAPPLICATIONID]
GED_NAME  [GED NAME]
END
```

The description of each field is shown below.

- BEGIN, END

These are used to denote the begin and the end of the configuration file

- MAPPING

This is used to map the old application identification and new application identification and vice versa in the GED server. Even though the application hasn't been ported to another machine, users still have to specify this mapping. The application ID is in the following format: "applicationName" and " _" and "hostName".

- GED_NAME

The GED_NAME is the name of the GED remote object. The value of GED_NAME in the Global.config has to be the same as the GED_NAME that is specified in the App.config. The applications use this name to lookup the GED remote object via the rmiregistry.

An example of Global.config is shown below. The application named "consumer" is ported to run on the machine named "tokyo" from the machine "newdelhi", whereas the application named “producer” still runs on the same machine. The GED server name is ged1.

```
BEGIN
MAPPING consumer_newdelhi consumer_tokyo
MAPPING producer_bangkok producer_bangkok
GED_NAME ged1
END
```

6.3 Implementation of Global Event Detector Initialization

In the global event detection, the GED server needs to be started before any other application. The GED server initially invokes the *initializeGECAgent( )* method inside the *GECAAgent* class. It begins with reading the given values from the *Global.config* file as described in last section, keeping the *GED_NAME* in the Constant class and mapping the information into the hash tables. Then, the global node event manager (*GlobalNodeManager*) object is instantiated to manage the global event nodes, which are created by the global event factory (*GlobalEventFactoryImp*). The *GlobalNodeManager* is responsible for maintaining these two hash tables. One hash table, which is called *glbEvntNm_GlbEvntNd*, stores the mapping between global event name and global event node. While the other (*glbEvntNm_GlbEvntHndle*) maps the global event names with the global event handles. Next, the GED server creates the communication interface (*GEDInterface*) that is used to send and receive messages. In addition, the server creates a *ServerConnectorImp* object for handling the client registration, a *GlobalEventFactoryImp* object for creating the global event node on the server, and a *GEDMesgRecvImp* object for handling the *Sentinel* messages, including *EventNotificationMessage* and *DetectionRequestMessage*. Then it binds these objects to the *rmiregistry* so that the clients can request for services. At this point the server is ready to support interaction with clients and global event detection.

6.4 Implementation of Local Event Detector Initialization

The application invokes *an initializeECAAgent( )* method of the *ECAAgent* class to initialize the LED. When the ECAAgent is initialized, the application configuration data are read from the file. If a scope is global, the extended part of LED is setup to accommodate the global event detection. A communication interface, a remote event factory, and a remote node manager are instantiated at this step.

The communication interface (*LEDInterface*), which is responsible for interfacing with the GED server, is initially created. When the *LEDInterface* is initialized, it binds the *LEDMesgRecvImp* object with the *APP_ID* and registers it to the *rmiregistry* so that the GED server can look up for the remote object (*LEDMesgRecvImp*) of the application, and make a remote invocation to send the message to this application. Then, it looks up for the *ServerConnectorImp* object in the registry. After obtaining this reference, it makes a remote invocation in order to register with the GED server. In addition, it also looks up for the *GlobalEventFactoryImp*, and *GEDMesgRecvImp* remote object; and keeps these references for the future use.

After initializing the communication interface, the remote event factory (*RemoteEventFactory*) and remote node manager (*RemoteNodeManager*) are created. For modularity purpose, these two classes are introduced to manipulate the remote event node. The *RemoteEventFactory* class is used to create the remote event node and construct the event graph. In addition, it helps in off-loading the *ECAAgent* class, since the users never create remote nodes directly and *ECAAgent* class mainly contains a set of APIs. It would be confusing for the users, if the methods of *RemoteEventFactory* class were mixed with the APIs in the *ECAAgent* class.

6.5 Implementation of Application's registration

As described in the previous section, the application registers with the GED server using an application ID when the agent is initialized. This application ID is a combination of an application name and a host name. The format of application ID is: *applicationName_hostName*. The application name is a given value from the application configuration file while the host name can be obtained at run-time. Each application can make a remote invocation to register with the GED server.

The GED server records the application ID and its address into the *clientAddrsHt* hash table. This address book (*clientAddrsHt*) is designed for the GED server to facilitate in looking up the client remote object in the rmiregistry. Since the GED server needs to know the host name in order to make a remote invocation to a particular client at a later time. The example of *clientAddressHt* is shown in figure 6.2.

clientAddressHt

Client application ID	IP Address
producer_bangkok	192.107.12.242
consumer_tykyo	197.107.12.241
[appName_machineName]	[appName_machineName]

Figure 6.2. An example of clientAddressHt (client address list).

The new application registers to the server in arbitrary order, and there might be other application that is already registered, since each application runs autonomously. Therefore, there might be a consumer application that has already registered and sent an event detection request associated with this new application. To determine whether there is an event associated with it or not, the GED server scans to the producer list that contains the producer ID and detection request messages. If it has an event detection request for the new application, the GED server will send the message back to the new application.

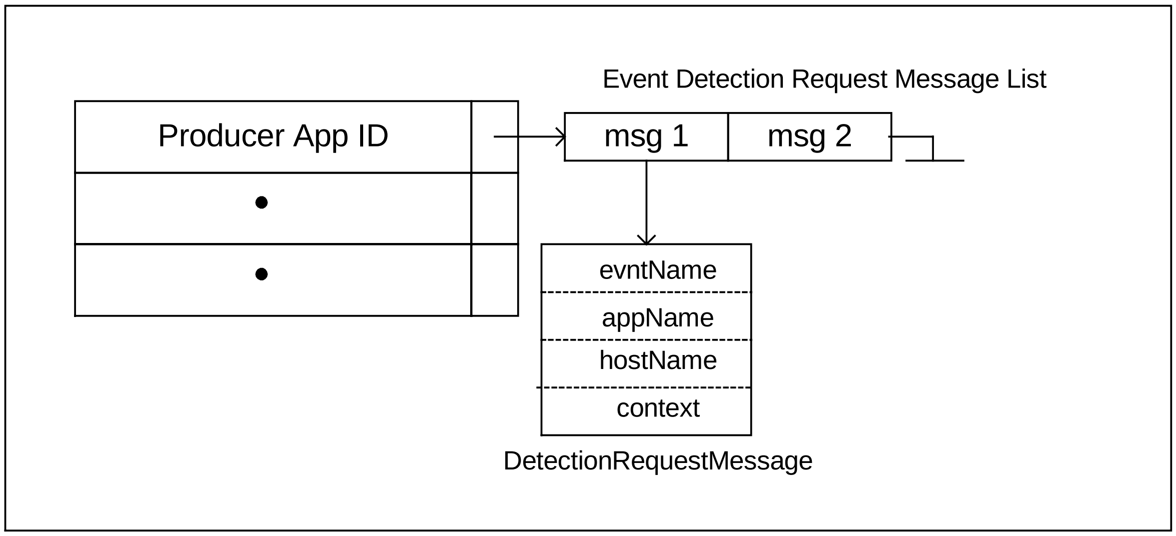


Figure 6.3. Producer Event List Data Structure.

As mentioned in the section, the producer list is used for keeping track of events to be detected and sent to the GED by a producer application. This producer list stores a producer ID or application ID, and the detection request messages corresponding to this application ID. When there is an event detection request from the consumer, the GED server inserts the detection request message, which contains the event information, into the producer list. Since there is typically more than one event defined in a producer application, there might be many detection requests corresponding to this producer. The GED server should use the application ID to retrieve the corresponding detection requests. Hence, a hash table is used for mapping between the application ID and the list of event detection requests in which requests are kept in a vector since it can grow dynamically. The producer list is shown in figure 6.3.

6.6 Implementation of Global Primitive Event Definition

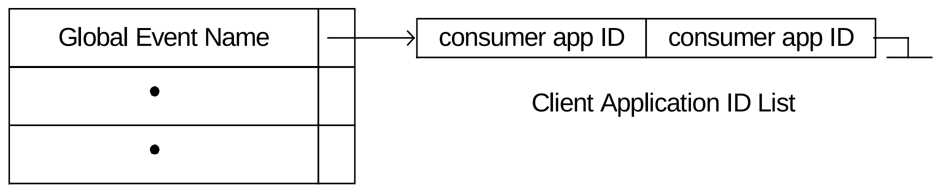
As a part of primitive global event definition, the application specifies the consumer event name that is known and referred by the consumer application, the full class name in which the global event is defined, the primitive event name that is defined and detected in the producer application, the producer application name, and the host name of the producer application. The following API is invoked within the ECAAgent instance to define a global event. The ECAAgent can be obtained at the initialization.

```
createPrimitiveEvent( String consEventName, String className, String prodEventName,
                      String appName,String machName)
```

When a local primitive event is defined, the LED constructs the primitive event node corresponding to that event. In contrast, the remote event node is created on the consumer application site when the global primitive event is defined. Then, the *LEDInterface* makes the remote invocation to create the global event node on the GED server. The details of how the server responds to this request will be discussed in the next section. After making the request, the *RemoteEventFactory* creates the remote event node, which represents the global event on the consumer site. This remote event node is always the leaf node of the local event graph. When the remote event node is created, a remote event handle corresponding to the remote event is returned. This handle can be used to create the global composite event. Next, the *RemoteNodeManger*, which is responsible for managing the remote event node, keeps the consumer event name and the remote event node in the hash table, which is called *consEvntNm_ConsEvntNd*. By using consumer event name, the system can retrieve the reference of the remote event node. However, the consumer event name is known and used only in the local site, the GED server knows nothing about this name. When the global event occurs, the server notifies the consumer application by sending the notification message containing the producer name, the application name and the host name. The consumer application needs to map this information back to the consumer event name so that it can reference back the event node. Hence, we define the global event name is a concatenation of the producer event name, application name, and the host name; and introduces *GlbEvntNm_ConsEvntNm* hash table to map the global event name with the consumer event name.

When the server receives an event detection request from a consumer application, the *GlobalEventFactoryImp* creates the global event node representing the global event on the server site if the global event node does not exist. Even when many applications are interested in the same global event, only one global event node is created. This global primitive event node is constructed as the leaf node of the global event graph. Similar to the *RemoteNodeManger*, the *GlobalNodeManager* maintains the global event node. It stores the global event name and the global event node into the *glbEvntNm_GlbEvntNd* hash table.

In addition, the GED server needs to update the consumer list of the corresponding event. By using the global event name, the server can look up for the associated consumer list and insert the application ID into it. Consequently, when the event occurs, the server will know which application has subscribed to this event. However, the event detection request will be sent to the producer application site when there is a rule applied on that global primitive event or the composite event that has that global primitive event as a constituent. The data structure of this consumer list is described in the next page.



- Global primitive name
  prodEvntNm+appNm+machNm
- Global composite name (binary operation)
  evntType+leftGlbPrimEvntNm+rightGlbPrimEvntNm
- Global composite name (ternary operation)
  evntType+leftGlbPrimEvntNm+ middleGlbPrimEvntNm +rightGlbPrimEvntNm

Figure 6.4. Consumer List Data Structure.

As shown in figure 6.4, the consumer list data structure is introduced to help the server keep track of the subscribers of each event. The server uses the event name to search for the applications that has subscribed to this event. Because of the above reasons, a hash table (*glbEvntName_consumerList*) and a vector are used for this data structure. The hash table maps the event name with the vector of application ID. When the new application wants to subscribe to a global event, the server adds its application ID at the end of the vector. When this event occurs, the server retrieves the application ID of consumers and uses that list to make a remote invocation to the client.

6.7 Implementation of Global Composite Event Definition

To define a global composite event, the application uses the existing set of APIs, which has been introduced in the LED. The application has to specify the event type such as *EventType.AND*, *EventType.OR*, or *EventType.NOT*, and a consumer event name that is used and known in the local application, and its constituents. We use the event handle to represent the constituent events. The set of APIs used for defining the global composite event is shown below.

```
CreateCompositeEvent (EventType eventType, String eventName, EventHandle leftEvent,
                      EventHandle rightEvent)
CreateCompositeEvent (EventType eventType, String eventName, EventHandle leftEvent,
                      EventHandle middleEvent, EventHandle rightEvent)
```

When the composite event is created, the system can distinguish whether this event is a global event or a local event by checking the *SCOPE* of event detector and the constituent events. If the SCOPE which is read from the application configuration file is global, and one of it constituents is a remote event handle; the system discerns that this is the global event. Then, it can determine where the event should be detected, and where the event node should be constructed (as described in an earlier Chapter). As we discussed in the last chapter, the location of the composite event node will be constructed at the GED server when all of it constituents are remote; otherwise, it will be constructed at the local site.

When one of the constituent events is detected at the local site or one of the constituents is not remote node; the global composite node is constructed in the same manner as when the local composite event is created. The composite event node stores the name of the composite event, and the references to the constituent event nodes. In addition, it prepares the *PCTable* for keeping the parameters of all constituent events. In this case, the LED is responsible for monitoring and detecting this event. Therefore, the *LEDInterface* doesn't need to send any request to the GED.

When a global primitive event is to be created at the GED server, , the *LEDInterface* sends a request to create a global composite event to the GED server. Since the composite event node will be created and detected at the remote site, the

RemoteEventFactory creates the remote event node after sending the request. This remote node, which is also a leaf node of the local event graph, represents the composite global event on the consumer site. In addition, the *RemoteNodeManger* also puts the event name and the reference of event node inside the hash table (*consEvntNm_ConsEvntNd*), and adds the global event name along with the consumer event name to the *glbEvntNm_GlbEvntNd* hash table. However, unlike the primitive global event name, the composite global event name is a concatenation of the name of the event type and the global event name of the constituent events. At the GED server site, *GlobalEventFactoryImp* constructs the composite event node after receiving the request. The *GlobalNodeManager* stores the global event name and the global event node into the *glbEvntNm_GlbEvntNd* hash table. In addition, the GED server also updates the consumer list of the corresponding event.

6.8 Implementation of Rule definition

Rule is comprised of a condition and an action, which can be implemented as methods in a class. To define a rule, the application specifies the rule name, the event handle of the event that is of interest, the condition method name and the action method name. By default, the rule is triggered when an e event occurs in the recent context. The application can define the rule to be triggered when the composite event occurs in a particular context (recent, chronicle, continuous, or cumulative). As mentioned in the previous chapter, there are four integers at each node. Each integer represents the sum of rules on that event and all the dependent events in a particular context. The corresponding integer will be incremented when a rule is applied on an event in a particular context. In addition, the corresponding integers of its constituent events from the event node till the leaf nodes along the tree hierarchy are also incremented. Hence, the composite event is detected only when either there is a rule associated with that event or a rule defined on another composite event for which this event is a constituent event.

```
createRule (String ruleName, EventHandle eventHandle,
            String condName, String actionName)
createRule (String ruleName, EventHandle eventHandle, String condName,
            String actionName, int priority, CouplingMode coupling,
            ParamContext context)
```

When a rule is associated with an event, the rule is simply inserted into the rule subscriber list and the context integers are incremented recursively on the tree rooted at the node where the rule was inserted. However, if the event is detected at the remote site, the *LEDInterface* needs to send the detection request message to the GED server. In other words, whenever the rule is defined as the global primitive event, the LEDInterface has to send the event detection request to the server. The global composite event might be detected at the consumer application site in some case; therefore, the *LEDInterface* doesn't need to send the request whenever the rule is applied to the global composite event.

In some cases, there are many rules defined on the remote event node with the same context. The corresponding integer representing the context information is also incremented proportional to the number of rules and all the dependent events. In the global event detection, however, the LEDInterface shouldn't send the event detection request to the GED every time when it is incremented except the first time. It will incur the communication overhead if the LEDInterface sends the event detection request whenever another rule is defined on the same event in the same context, since it will send the same information to the GED server. However, if the rule is defined on the global event in a different context, the LEDInterface will send the event detection request message containing the new context information to the GED server so that it can detect the event in the new context.

Once the GED server receives the event detection request message, the GED server will set the *sendBack* flag on the corresponding global event node. The GED server will send the notification message to the subscribers or consumers only when the global event is detected on the server and the sendBack flag is true, since some global events are defined as the constituents of the composite event, and there is no rule applied on these constituents. In addition, the GED server needs to increment the context value at the corresponding node in the same manner as incrementing the context value of the remote node at the consumer site. The only difference is that it will send the event detection request to the producer site when the context value at the leaf node of the global event graph is equal to one.

When the producer application receives the event detection request, the LED will set the *forwardFlag* of the corresponding node to be true and also increment its the context. So that when the event occurs in that context, the *LEDInterface* will notify to the GED server. The details of the event detection will be discussed in the next section.

6.9 Implementation of Global Primitive Event Detection

Any method in the Java application can be defined as a primitive event. For example, the *setTempature( )* is defined as a primitive event in the weather application. When this method is invoked, the event detection mechanism should be able to detect the occurrence of event and notify to all its subscribers. In order to detect the primitive event, the event detector requires the user to signal the occurrence of the event by using *raiseBeginEvent* and *raiseEndEvent* API calls. The arguments to the method need to be inserted into the event handle, which is passed through these APIs. Since the event handle stores the signature of the method, the event detector can use it to retrieve the corresponding primitive node from the *eventSignsEventNodes* hash table. Then, the node is notified about the occurrence of this event and the rules associated with this event are triggered (if any). For the global event detection, the *forwardFlag* flag in each node is checked whenever it is notified. If the *forwardFlag* is true, it means that this event is subscribed by another application in the different address space. The *LEDInterface*, which is responsible for notifying the GED server, prepares the event notification message and sends it to the server. To send the message, the *LEDInterface* looks up the server remote object registered in the *rmiregistry*, and makes a remote invocation to the remote object. As mentioned earlier, the notification message contains the event name, application name, hose name, and *PCTable*. When the primitive event occurs, the parameter associated to this event is encapsulated into the PCTable. Even though there is only one parameter list in the PCTable, we don't put the *ParameterList* object instead of *PCTable* inside the message, since we want to make the event notification message format the same for both global primitive events and global composite events.

After receiving the notification message, the GED concatenates the event name, producer application name and host name to make global event name. Then, it uses this name to retrieve the corresponding global event node from the *glbEvntNm_GlbEvntNd* hash table. When it notifies the occurrence of this global event node, it checks the *sendBack* flag to determine whether it needs to send the notification message back to the consumer application or not. If the flag is set, the GED server searches for the consumers of this event in the *eventName_consumerList*, and forwards the notification message to each consumer.

When the GED server notifies the consumer application, the *LEDInterface* receives the notification message containing global event information and the parameter lists. As we mentioned before, the consumer event name is used in the local site instead of the global event name. Therefore, the *LEDInterface* needs to translate the global event name into the consumer event name, which is used in retrieving the remote event handle in the *remEvntNm_RemEvntHndle* hash table. Similar to the local primitive event detection, the remote event handle is used to instantiate the *NotifyObject* so that the *LEDInterface* can put this *NotifyObject* into the *NotifyBuffer*. Running separately in a finite loop, the local event detector thread gets this object from the buffer and invokes the method calls with the parameters stored in the object as if it detects the local primitive event in the previous version of the local event detection.

6.10 Implementation of Global Composite Event Detection

As mentioned earlier, the local composite event is composed of local primitive events and/or other local composite events by applying event operators, and it can be detected at the local site by the local event detector. However, a global composite event can be detected at either local site or server site. Checking the constituent events, the system can create the global composite event node and detect it at the appropriate place. If at least one of the constituent events is a global event (global primitive or global composite event), the LED at the consumer site is responsible to detect this composite event. In contrast, if all of the constituent events are global events, the global composite event is detected at the server.

The approach to detect the global composite event that at the local site is similar to that of the local composite event, but at least one of the occurrences of the constituent events including its descendents is signaled from the GED server.

In order to detect a global composite event on the server, we also use the event graph at the server. Each composite event has an initiator and a terminator event. When its constituent event is detected, it will propagate a list of parameter lists to all the event nodes that have subscribed to this event, when it notifies the occurrence of the constituent event. When the terminator event of the composite is detected, the composite event will be detected and notified. And if there is a rule defined on the composite event or rule defined on another composite event for which this composite event is a constituent, this composite event will be detected in that particular context. Similar to a global primitive event, when the global composite event is detected on the server, it will check the *forwardFlag* whether it needs to send the notification message to the corresponding consumers or not. It should be noted that a remote node is also used to represent the global composite event in the consumer site. Hence, when the *LEDInterface* receives the notification message form the GED server, it will notify the remote event node with the list of parameter lists, and it continues notifying all the event nodes that are in the event subscriber list, and propagating the list of parameter lists. All the rules associated with this event are also executed.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

This thesis extends earlier work on the Local Event Detector (LED) implemented in Java and presents an approach to monitor events in a distributed environment. The Local Event Detector was designed and developed to provide support for events and rules in a seamless manner in the Java application environment. It is used to monitor event and respond to the changes within a single address space. Since a large number of applications today are distributed in nature, our implementation of Global Event Detector (GED), based on the notification/subscription paradigm, uses the ECA (Event-Condition-Action) rule paradigm in order to support active capability in a distributed environment.

In this implementation, the existing Local Event Detector has been extended to accommodate global event detection involving one or more applications, and a GED as a server is introduced to provide the subscription and notification services in a distributed environment.

A summary of the each chapter is provided.

Chapter 1 explains the motivation and defines the problem. It also explains why the earlier work is not adequate to support and monitor events in a distributed application environment.

Chapter 2 describes recent research work on a distributed event-based system and related work on event services and event notification mechanisms.

Chapter 3 discusses the semantics of primitive t and composite events.

Chapter 4 provides a summary of the Local Event Detector in terms of event specification and the data structures used.

Chapter 5 describes the design of the Global Event Detector. It discusses alternative architectures, communication mechanisms, optimization techniques to minimize the computation cost and communication.

The implementation of the Global Event Detector is described in chapter 6. The global event definition and rule definition are introduced for defining ECA rules. The event subscription and event detection mechanisms are described.

7.2 Future Work

A number of extensions are planned beyond the current implementation.

Robustness of the Global Event Detector and event persistence issues are not addressed in the current implementation of the GED. It is important to address these issues since rules can be specified on events that occur in one or more applications, and there should be no surprise when a failure occurs, as the distribution of events is not sufficient to make the distributed application reliable. Global Event Detector should be able to recover to a consistent state following various types of failures, and continue to provide services after it recovers. It should also be possible to tolerate client crashes and provide loss less delivery of events.

Guaranteed delivery of events: It is essential to ensure the delivery of notification messages or subscription messages since messages can be lost due to the communication failure.

Implementing publish/subscribe paradigm: This paradigm is useful when the consumers are concerned more about the event topic rather than the source of event. The system should allow the consumers to subscribe to the event without the knowledge of the source.

Supporting content based filtering: In this implementation, the event can be filtered out at the subscriber by using ECA rule mechanism. It would be interesting to filter events based on its content at the server before notifying the occurrence of an event to its subscribers. This will further minimize event traffic and network load.

Scalability and performance issues need to be further investigated. Also, the possibility of multiple GEDs for replication and availability has not yet been addressed.

REFERENCES

1. http://java.sun.com/beans/infobus/, *InfoBus*. 1999.

2. Object Management, G., *{CORBAServices: Common Object Services Specification v1.0}*. 1995: John Wiley \& Sons Inc. NJ.

3. Dasari, R., *Events And Rules For JAVA: Design And Implemenation Of A Seamless Approach*, in *Database Systems R&D Center, CIS Department*. 1999, University of Florida: Gainesville.

4. Schmidt, D.C. and S. Vinoski, *The OMG Events Service. C++ Report*. 1997.

5. http://msdn.microsoft.com/library/en-us/cossdk/htm/pgservices_events_20rp.asp?frame=true, *COM+ Events Architecture*. 2001.

6. Scarlett, S., *Monitoring the Behaviour of Distributed Systems*, in *Cambrigde University Computer Laboratory*. 1996, University of London: London.

7. Liao, H., *Global Events in Sentinel: Design and Implementation of a Global Event Detector*, in *MS Thesis*. 1997, Database Systems R&D Center CISE University of Florida, Gainesville, FL 32611.

8. Chakravathy, S. and D. Mishra, *An Event Specification Language (Snoop) for Active Databases and its Detection*. 1991, Database Systems R\&D Center CIS Department University of Florida.

9. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases.* Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.

10. Chakravarthy, S., et al., *Composite Events for Active Databases: Semantics, Contexts and Detection*, in *Proc. Int'l. Conf. on Very Large Data Bases VLDB*. 1994: Santiago, Chile. p. 606--617.

11. Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Thesis*. 1994, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611.

12. Stonebraker, M., L. Rowe, and M. Hirohama, *The Implementation of {POSTGRES}*. IEEE Transactions on Knowledge and Data Engineering, 1990. **2**(1): p. 125--142.

13. Song, Z., *A Generalized Approach For Extending The Active Capability Of RDBMSs*, in *Database Systems R&D Center, CISE Department*. 2000, University of Florida: Gainesville.

14. Kim, Y., *A Generalized Active Agent System For Extending The Active Capabilities Of A RDBMS*, in *Database Systems R&D Center, CISE Department*. 2000, University of Florida: Gainesville.

15. Vondrak, C., *Message-Oriented Middleware*. 1997.

16. http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp, *DCOM Architecture*. 1997.

17. http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html, *Remote Method Invocation Specification*. 1999.

18. David E. Culler, J.P.S., Anoop Gupta, *Parallel Computer Architecture : A Hardware/Software Approach*. 1st Edition ed. 1998. 1100.

BIOGRAPHICAL SKETCH

Weera Tanpisuth was born on August 2, 1975 in Bangkok, Thailand. He received his Bachelor of Science degree in Electrical Engineering from Thammasat University, Thailand in March 1997, and Master of Science degree in Electrical and Computer Engineering from University of Florida in August 2000. In the Fall of 2000, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science from The University of Texas at Arlington, in December 2001. His research interests include active and object-oriented databases.