RUBCES : A RULE BASED COMPOSITE EVENT SYSTEM

Ozgur K. SAHINGOZ
Air Force Academy
Computer Engineering Department
Yesilyurt, Istanbul, TURKEY
e-mail: o.sahingoz@hho.edu.tr

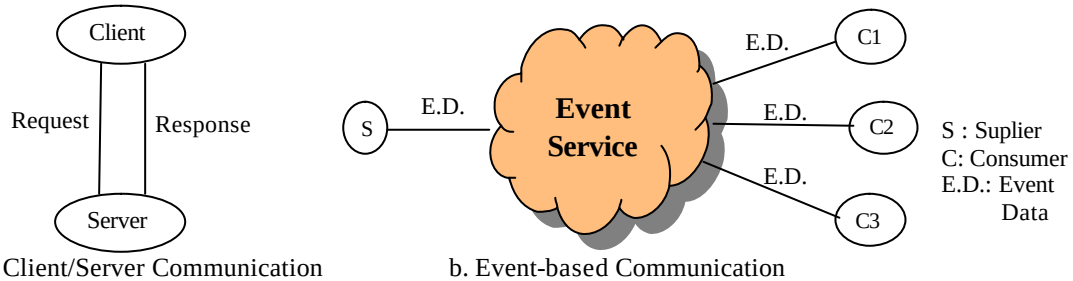
Nadia ERDOGAN
Istanbul Technical University
Electrical-Electronics Faculty
Computer Engineering Department
Ayazaga, 80626, Istanbul, TURKEY
e-mail: erdogan@cs.itu.edu.tr

Abstract: An event is the occurrence of some state change in a component of a software system, made visible to the external world. Usually an event is represented by a data structure called an event notification, or simply a notification. Sometimes subscribers can express interest in being notified upon the occurrence of specific combinations of events only. This type of an event is called a composite event. RUBCES (Rule Based Composite Event System) is a centralized event system that allows the use of composite events in publish/subscribe computational model. In this system, an event is represented as an object and a rule is represented as an expression or a function that is evaluated or executed depending on the occurrence of events. This paper presents the design details of RUBCES and the syntax and semantics of Rule Definition Language (RDL) for describing composite events and event filters.

Keywords: event system, composite events, rule based events, event filters.

1. Introduction

In the traditional client/server computing model, which is used in RPC and RMI, communication is typically synchronous, tightly coupled and point to point. As shown in Figure 1.a, clients invoke a method on the remote server and wait for the response to return. This type of communication requires clients and servers to have some prior knowledge of each other.



a. Client/Server Communication b. Event-based Communication

Figure 1: Client/Server and Event-based Computing Models

With the use of mobile or large-scale systems, the need for asynchronous, loosely coupled and point to multipoint communication pattern arises. Event models are application independent infrastructures that satisfy communication requirements of such systems. Event-based communication generally implements what is commonly known as the publish/subscribe protocol. As shown in Figure 1.b, an *event supplier* asynchronously communicates *event data* to a group of *event consumers*, ideally without knowledge of the number and location of the event consumers.

To receive event data, consumers register to an event service with the definition of the particular events they are interested in. This definition can include a simple subscription message, a subscription message with filtering on events or a subscription

message for composite events. This study is on an event-based system that uses a new language, Rule Definition Language (RDL), to define subscription criteria in the form of a rule.

The rest of this paper is organized as follows. In the next section, we present a classification of publish/subscribe systems, with references to related work. Section 3 introduces the computational model of RUBCES with RDL syntax and semantics. Section 4 focuses on the event specification and event types. Our conclusions and plans for future work are presented in Section 5.

2. Publish/Subscribe systems

Publish/subscribe programming paradigm is characterized by the complete decoupling of producers (publishers) and consumers (subscribers). The event service provides message transfers between publishers and subscribers can be decomposed along the three following dimensions.

Time decoupling: the interacting parties do not need to be up at the same time.

Space decoupling: the interacting parties do not need to know each other.

Flow decoupling: producers are not blocked while producing events and consumers can get notified about the occurrence of some event while performing some concurrent activity.

Publish/subscribe systems can be classified into four groups according to their subscription mechanism. Each one is discussed in detail below, with references to representative work.

2.1. Channel-based subscriptions

The simplest subscription mechanism is what is commonly referred to as a *channel*. Subscribers subscribe or listen to a channel. Applications explicitly notify the occurrence of events by posting notification to one or more channels. The part of an event that is visible to the event service is the identifier of the channel to which the event has been sent. Every notification posted to a channel is delivered by the event service to all the subscribers that are listening to that channel.

The abstraction of the channel is equivalent to the one given by a mailing list. A user sends an e-mail to an address, and message is forwarded to those who have registered to that mailing list. CORBA Event Service [1] adopts a channel-based architecture. Another widely used channel based model is the Java Delegation Event Model [2], which encapsulates events from the platform's Graphical User Interface. It consists of three main classes objects:

- Source Object - generates an Event Object in response to a (user) occurrence and passes the event object to concerned listeners.
- Event Object - is passed from the Source Object to the listeners. The Event Object contains information describing the occurrence.
- Listener Object - receives the Event Object via a message and performs appropriate processing depending upon the details of the event object.

2.2. Subject-based subscription

Some systems extend the concept of a channel with a more flexible addressing mechanism that is often referred to as *subject-based* addressing. In this case, an event

notification consists of two different parts: a well-known attribute, *the subject*, that determines their address which is followed by *the remaining information* of the event data. The main difference with respect to a channel is that subscriptions can express interest in many subjects/channels by specifying some form of expression to be evaluated against the subject of a notification. This implies that a subscription may define a set of event notifications, and two subscriptions may specify two overlapping sets of notifications. This, in turn, implies that one event may match any number of subscriptions.

JEDI [3] adopts the subject-based subscription mechanism. In JEDI, an event is given in the form of a function call; where the first string is the function/event name followed by parameters, e.g., "print (tez.doc, myprinter)". Each event is labeled with a subject. Subscriptions are specified with an indication of the subject of interest. Notice that the subject-based approach is a variation of the channel based concept, as the rest of the event data except for the subject is content-free. The subject can be a list of strings in a hierarchical form, over which it is possible to specify filters based on a limited form of regular expressions. For example, the filter "economy.exchange.*.*HOL" will select all the notifications whose subject contains economy in first position followed by exchange in second position, any string in third position, and a fourth string that ends with the string "HOL".

2.3. Content-based subscription

By extending the domain of filters to the whole content of notifications, some researchers obtain another class of subscriptions called content-based [5]. Content-based subscriptions are conceptually very similar to subject-based ones. However, since they can access the whole structured content of notifications, an event server gives more freedom in encoding the data upon which filters can be applied and that the event service can use for setting up routing information.

```
string event = accnt/bbb
time date = 15.11.2002
int num = 12345
float total = 215.31
```

a. SIENA Event Data

```
string event == accnt/*
time date >= 01.01.2000
float total > 100.000
```

b. SIENA Filter

```
string event > finance/*
string symbol = SAHOL
float change < 0
       and then
string event > finance/*
string symbol = DOHOL
float change > 0
```

c. SIENA Composite Event

Figure 2: SIENA Event and Filter Models

Examples of event systems that provide this kind of subscription are, Yeast [4](this is also using centralized structure) and SIENA[5]. In SIENA, an event notification is a set of attributes, as shown in Figure 2.a, in which each attribute is a triple, as in "attribute = (name; type; value)". Attributes are uniquely identified by their name. An event filter, as shown in Figure 2.b, defines a class of event notifications by specifying a set of attribute names and types and some constraints on their values, e.g., "attr filter = (name; type; operator; value)". A composite event is defined by combining a set of event filters using filter combinators as shown in Figure 2.c.

2.4. Object-based subscription

Object-based subscription is a new model of subscription that has been developed to access event data in a more structured manner. Type-based publish/subscribe mechanism [6], proposed by Eugster, uses object events, which are called *obvents.* Events are often viewed as low-level messages, and a predefined set of such message types are offered by most systems, providing very little flexibility. To overcome this deficiency, type-based publish/subscribe mechanism manipulates events as objects. The core idea underlying this integration consists in viewing events as first class citizens, and subscribing to these events by explicitly specifying their type. So an application defined event data can be used in the event system.

```
publish o;
```

```
Subscription s = subscribe (StockQuote q)
             // First Block
{    return (q.getPrice() < 100 &&
      q.getCompany().indexOf("Telco") != -1);  }
             // Second Block
{    System.out.print("Got offer: ");
     Sytem.out.println(q.getPrice());   };
```

Figure 3: Primitives of Obvent System

Obvent system uses two basic primitives as shown in Figure 3. An obvent "o" is published through a primitive publish, and a subscriber registers to the event system by using the subscribe syntax which contains filter constraints in the first block, and declare triggering events in the second block.

3. RUBCES Computational Model

RUBCES is an event-based publish/subscribe system that uses rules for subscribing to an event service. Many of the event systems described in literature, as referenced in the previous section, use predefined events. RUBCES implements a content based subscription mechanism, similar to that proposed by Carzaniga [5], which allows handling of application-defined events. Our event system also provides mobility for subscribers. By using an "id" and "password" pair, a subscriber can connect to the Event Server from different machines.

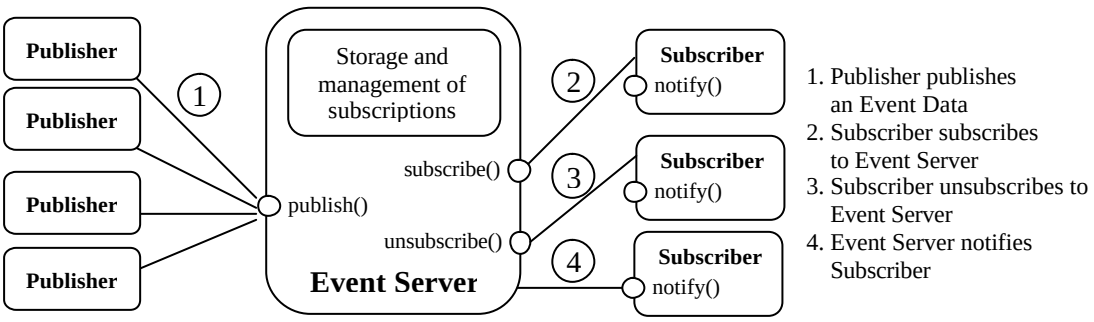


Figure 4: RUBCES Architecture

RUBCES, being implemented in Java, makes use of the Java RMI facilities extensively to access remote objects. To create a uniform structure, the components of the system are designed to be accessed over well-defined interfaces and, are expected to

implement the methods included in those interfaces. Figure 4. depicts the general architecture of RUBCES. The system consists of three main components: an Event Server, subscribers, and publishers.

3.1 Event Server

The main function of the Event Server is dispatching incoming event notifications from publishers to (possibly multiple) subscribers. The event server implements the *EventServer* interface, shown in Figure 5, which also extends the *Remote* class to enable RMI. The *EventServer* interface consists of three main methods:

```
public interface EventServer extends Remote
{
public void subscribe(String codebase, String rule, String sub_name,
                String subs_pass, String Store_type) throws RemoteException;
public void unsubscribe( String codebase, String rule String sub_name,
                        String subs_pass) throws RemoteException;
public void publish(Event e, String class_address) throws RemoteException;
}
```

Figure 5: Java definition of the EventServer Interface

subscribe: A subscriber registers interest in a particular event by invoking the *subscribe* method of the event server. It supplies its RMI contact address and a rule that describes the events it is interested in as parameters to the call. The system supports mobile subscribers by allowing them to join the system from different locations, after establishing a connection through a name/password pair.

unsubscribe: A subscriber can cancel its registration through a call to the *unsubscribe* method, supplying parameters needed to identify the subscription previously made. For security reasons subscriber has to provide its subscriber name and password for the unsubscribe operation to proceed successfully.

publish: Publishers call the *publish* method to announce an event.

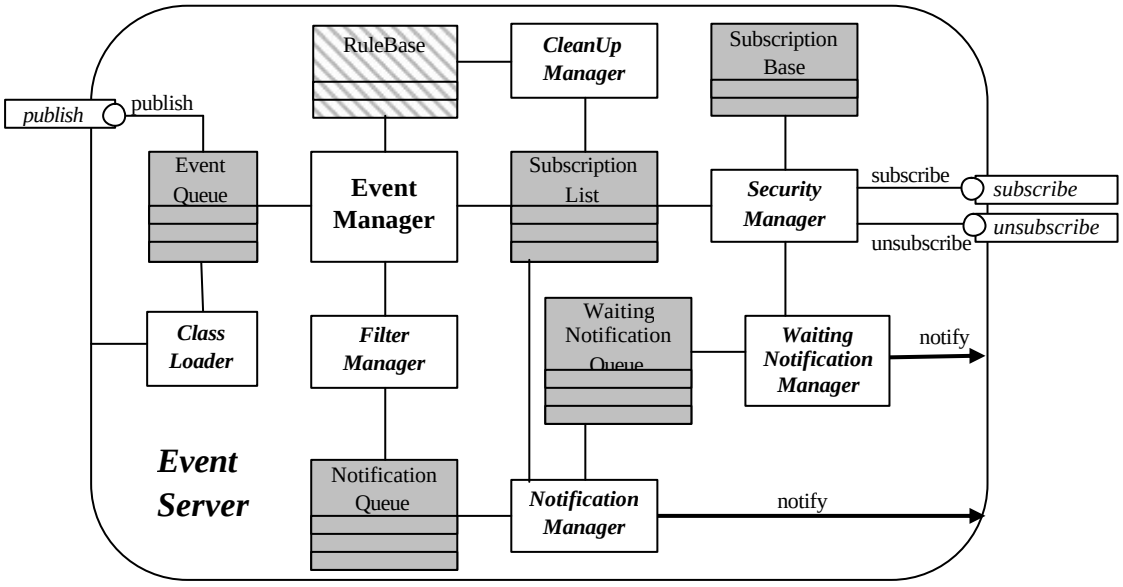


Figure 6: The Architecture of the Event Server

The Event Server consists of manager modules that handle the basic functionalities of the system and several structures to hold various data. The internal architecture of the Event Server and message flow between its components are shown in Figure 6. In the following, we first describe the main data structures of the Event Server, and then focus on the manager modules.

Event Queue: *Event Queue,* where an incoming event data is initially stored, is a vector which has the fields of (Event_type, Serialized_data, Class_addr, dowloaded).

Subscription List: When a subscriber registers to the event system, its registering information is stored in the *Subscription List* in the format (Event_type, Subscriber_adress, Subscription_type, Triggering_SQL, Store_type).

Subscription_type: A subscription may be of three types, depending on the type of event it involves, either a simple event, a filtered event, or a composite event.

Store_type: This information specifies whether a subscriber wishes notification data be stored or not in the system while it is in unconnected state. A subscriber may require either all or only the last notification data to be stored or it may not be interested in such data at all.

Triggering_SQL: defines SQL commands which are to be triggered when an incoming event is related to a composite event.

RuleBase: Some subscribers may want to detect a specific pattern of event occurrences from different publishers (composite events). To catch a specified sequence of events, the system stores events which have previously occurred in the *RuleBase*. Naturally, not all events, but only those that are part of a composite event are stored in the *RuleBase*. *RuleBase* is designed as a database that includes tables representing the incoming events. For each different event type, a new table is created by the *Event Manager.* Connection to the *RuleBase* is established through Java Database Connectivity (JDBC) and data is exchanged by dynamically produced SQL commands..

Subscriber Base: This is where subscriber names and passwords are stored. The security manager prevents unauthorized access using the data. *Subscriber Base* is designed as a vector which has the fields (Subscriber_name, Subscriber_password).

Notification Queue: An outgoing event data is stored in the *Notification Queue* before it is used by the *Notification Manager*. The queue is implemented as a vector which has the fields of (Event-type, Subscriber_name, Subscribers_address, Serialized_data).

Waiting Notification Queue: If the *Notification Manager* cannot get in contact with a subscriber, it stores the unsent notification data in *Waiting Notification Queue*, if required so by the store-type parameter of subscription. The *Waiting Notification Queue* is similar in structure to the *Notification Queue.*

Several modules use data in the three queue structures listed above, as depicted in Figure 6. To prevent any inconsistency, mutually exclusive access is provided through synchronized methods.

Class Loader: Downloads classes related to incoming events to the Event Server's codebase.

CleanUp Manager: When subscribers unsubscribe from the *Event Server*, CleanUp Manager deletes their registration information from the *RuleBase*.

Security Manager: Two functionalities of the security manager are the following.

1. Monitors entrance of subscribers and prevents unauthorized access.
2. A subscriber can connect to the event server from different locations (addresses). If a subscriber reconnects after losing contact, the Security Manager recognizes this condition and activates the *Waiting Notification Manager* to send event data in the *Waiting Notification Queue*, if there is any.

Event Manager: This is the main processing module of the *Event Server*. When a new event (EventX) arrives, the Events Manager follows the following algorithm to process it:

1. It controls the *Subscription List* to detect any subscription that contains EventX
2. If it finds such a subscription, then it determines whether it is a composite event subscription or not.
3. If it is not a composite event subscription, it forwards the event data and subscription record to the *Filter Manager.*
4. If EventX takes part in a composite event,
    a. *Event Manager* runs the *Triggering_SQL* in the *Subscription List,* gets the query results and forwards these results together with event data and the subscription record to the *Filter Manager.*
    b. If (SQL) table of EventX has been created beforehand, then *Event Manager* adds event data to that table. Otherwise, it first creates a new table for EventX, and then adds event data to that table.

Filter Manager: Carries out actions to see if incoming events, or those stored in the *RuleBase* satisfy the filtering conditions of any subscription. If one is found, then the related event data is entered the *Notification Queue*.

Notification Manager: Sends notifications that are placed into the *Notification Queue* to the target subscriber. If it cannot get in contact with the subscriber, it stores that particular notification data in the *Waiting Notification Queue,* if later notification is required so by the subscriber.

Waiting Notification Manager: If a subscriber reconnects to the system and requests its waiting notification messages, this manager sends the stored notifications to the subscriber.

3.2 Subscriber

Subscribers of events determine what types of information they are interested in and describe them in a form usable by the Event Server. Subscribers have to implement the *Subscriber* interface, shown in Figure 7, which consists of single method, *notify* that extends the *Remote* class to enable RMI. The event server issues a remote call to the *notify* method of the subscriber to deliver an event. The subscriber is expected to process the event in the context of this method.

```
public interface Subscriber extends Remote
{
    public void notify(Event event[]) throws RemoteException;
}
```

Figure 7: Java definition of the Subscriber Interface

3.3 Publisher

Publishers of information decide on what events are observable, how to name or describe those events, how to actually observe the event, and then how to represent the event as a discrete entity. A publisher process is not required to implement a particular interface. The Event Server address has to be known by the publisher so that it can issue a remote call to the `publish` method of the Event Server.

3.4. Rule Definition Language (RDL)

A rule is an expression or function that is evaluated or executed depending on the occurrence of events. We have developed a language, Rule Definition Language (RDL), to state rules to aid the specification of a single or a pattern of events in distributed systems. The grammar of the language is presented in Figure 8, in BNF notation, with highlighted keywords.

```
<Rule_def> ::=   <rule>| <rule> where <Condition>;
<rule>     ::=   rule identifier; onEvent <Events>;
<Events>   ::=   class/interface_type identifier |
                 class/interface_type identifier , <Events> ;
<Condition>::=   Condition <Boolean_Operator> Condition |
                 (Condition) |! Condition | <Exp> <Relation_Operator> Exp>
                 | true |  false
<Exp>      ::=   ( <Exp> ) |  <Exp> Arith_Operator <Exp> | identifier
<Arith_Operator>   ::=    + | - | * | /
<Relation_Operator>::=    > | < | >= | <= | == |!=
<Boolean_Operator> ::=    and | or
```

Figure 8: RDL Grammar

4. Event Specification and Event Types

The class Event is the base class of all event classes. It defines common attributes for all types of events. Figure 9 presents the Java definition of the class Event.

```
public abstract class Event implements Serializable{
      /******* Event attributes *********/
    protected long time;                // production time
    protected String eventType;         // name of the event
    protected int sequenceNo;           // sequence number of the event
      /******* Event methods *********/
    public Event()                      //constructor of the event without parameter
    public Event(long time_, String eventType_, int sequenceNo_)
                                        // constructor of the event with parameter
    public long getTime()               //return the production time
    public void setTime(long time_)      //set the production time
    public String getEventType()         //return the event name
    public void setEventType(String eventType_)      //set the event name
    public int getSequenceNo()                      //return sequence number
    public void setSequenceNo(int sequenceNo_){}     //set sequence number
        public String toString()                    //convert to string expression
      }
```

Figure 9: Java definition of the Event Class

The attribute *eventType* represents a unique identifier for each event object, while the *time* attribute defines the time, relative to the day, when the event is produced. The *sequenceNo* attribute is used to control repeating event messages.

A subscriber specifies the type of event that describes which event it wishes to be notified of. The class definition of the event extends the base class *Event* and may contain additional structures. Figure 10 shows the specification of a sample event type class named *HeatEvent*. In addition to the attributed inherited from the *Event* class, this class defines a new attribute *loc* to identify the location of the event produced and *value* to represent a measure of some type, for example, temperature reading of a thermometer.

```
public class HeatEvent extends Event implements Serializable
{
       int loc;       // location of the event
       double value;  // measurement of the temperature
}
```

Figure 10: An Example of an Event Class

A rule definition is composed of three main parts: the first part sets a unique identifier for the rule, in the second part the type of event is specified, and in the third part the requested criteria is declared, in case a filtered or a composite event is being defined. In RUBCES, it is possible to use rules to define three different event types: simple events, events with filtering and composite events. In simple events, shown in Figure 11, subscribers are interested with only one event type.

```
rule rule1                    | rule rule2
onEvent HeatEvent h1;         | onEvent StockExchangeEvent s1;
```

Figure 11: Simple Events in RDL

An event-based system may consist of a number of publishers, all of which produce events that may contain different information. Thus, the number of events propagated in an event-based system may be quite large. However, a particular consumer may only be interested in a subset of the events propagated in the system. Event filters are a means to control the propagation of events. Filters enable a particular consumer to subscribe to the exact set of events it is interested in receiving. An event that is delivered uses network bandwidth and CPU processing power on the consumer side. It is therefore desirable to prevent the delivery of unwanted events. In RUBCES filtered events are used as shown in Figure 12.

```
rule rule3                    |   rule rule4
onEvent HeatEvent h1;         |   onEvent HeatEvent h1;
where h1.value > 25;          |   where (h1.value > 25 and
                              |      h1.value < 37);
```

Figure 12: Filtered Events in RDL

Clients may require to be notified on events from multiple sources and may want to detect a specific pattern of event occurrences from these different publishers. Such a combination of event occurrences, where a client is interested in a sequence of event occurrences but not in any of the events alone, is called an *event composition*.

Intuitively, while a filter selects one event notification at a time, a pattern can select several notifications that together match an algebraic combination of filters. Composite events are used in RUBCES as shown in Figure 13.

```
rule rule5
onEvent Temperature t1,
        Humidity h1
where (t1.value < 27 and
       h1.value < 70)
```

```
rule rule6
onEvent HeatEvent h1, HeatEvent h2,
        LightEvent l1;
where
   (h1.value > h2.value + 5)
   and (h1.time > h2.time+5000)
   and l1.lightvalue> 3000);
```

Figure 13: Composite Event Subscription in RDL

5. Conclusions

In this paper, we have presented RUBCES, an event-based publish/subscribe system that uses rules to specify events. RUBCES uses a content-based subscription mechanism that allows user defined event types and allows for subscriber mobility. A new Rule Definition Language (RDL) is developed and used to specify different types of events (simple subscription, subscription with filtering and subscription for composite events). Currently, a prototype system in Java is being implemented. As future work, we have plans to apply the system in different application domains and focus on new design decisions to improve its scalability.

References

1. Object Management Group, “CORBAservices: Common Object Service Specification”, Technical Report, Object Management Group, July 1998.
2. “Java AWT: Delegation Event Model”. Available online at http://java.sun.com/j2se/1.4.1/docs/guide/awt/1.3/designspec/events.html
3. G. Cugola, E. Di Nitto, and A. Fuggetta, “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS”, Technical Report, CEFRIEL - Politecnico di Milano, Italy, August 1998.
4. B. Krishnamurthy and D. S. Rosenblum, “Yeast: A General Purpose Event-Action System”, IEEE Transactions on Software Engineering, 21(10):845–857, Oct. 1995.
5. A. Carzaniga, “Architectures for an Event Notification Service Scalable to Wide-area Networks”, PhD Thesis, Politecnico di Milano, Italy, December 1998.
6. P.T.Eusgter, “TypeBased Publish/Subscribe”, PhD Thesis. Ecole Polytechnique Federale De Lausanne, France, 2001