Ubiquitous Computing
University of Illinois at Urbana-Champaign
Computer Science Department

Bhaskar Borthakur
Software Research Group
October 2001

Request for Comments: 0003
Obsoletes: none

The Event Service

Table of Contents

1.0 Status of this Memo

The memo is in the request for comments state. This memo will undergo changes as and when required.

2.0 Introduction

Components in a loosely coupled system are typically designed to operate by generating and responding to asynchronous events. The event manager provides a model for decoupled communication between different entities in Gaia. The event manager uses the underlying event service of Orbacus-Corba for communication. In our current implementation Event Manager exports an interface for both pull and push based event delivery. It allows creating channel categories, browsing these categories and their associated channels and creating and deleting channels associated with particular categories. All Gaia components use the event manager to learn about changes in the state of the space and react accordingly. The discovery service, for example, listens to different channels to learn about new entities (e.g. software services and people entering the space), filters the information, and publishes events to inform the rest of the system about new entities discovered and entities no longer available.

3.0 Requirements

3.1 Fault tolerance

In a distributed event system, if any of the event service or event channels crashes there should be a mechanism for the clients to reestablish the channel / service. When the event manager recovers after a crash it should be possible to regain its initial state.

3.2 Persistence

Events need to be persistent. There should be a well-known (or discoverable) repository of events. In order to prevent an explosion of events, the archived events should have a time to live field.

3.3 Content Based Query Processing

It should be possible to look up past (within a bounded interval of time) events based on the contents of the events. This implies that there should be a facility to query for events with particular values in a type-value pair or the name of the source or the destination.

3.4 Low Latency

For events like mouse and keyboard, the event service needs to provide a means of fast communication. In an active room having multiple displays, mouse and keyboard events need to be redirected to the displays. Typically there will be a single sender and single receiver of events in such a scenario. The preferred rate of event delivery should be around 60 events per second with a payload of around 50 bytes.

3.5 Federation

It should be possible to federate event services of different active spaces to incorporate the needs of mobile users. A user who subscribes to a particular channel in a particular active space should get his events forwarded when he moves to a new space.

3.6 Secure Channels

Communication through events should be secure and authentic.

4.0 Features and Components in the Event System

4.1 Suppliers and Consumers

The Corba event service defines two roles for objects: the supplier role and the consumer role. *Suppliers* produce event data and *consumers* process event data. Event data are communicated between *suppliers* and *consumers* by issuing standard CORBA requests.

4.2 Push and Pull Model

There are two approaches to initiating event communication: *push model* and the *pull model*. The push model allows a supplier of events to initiate the transfer of event data to consumers. The pull mode allows consumers of events to requests events from a supplier. The consumer is taking the initiative in the pull model while in the push model the supplier is taking the initiative.

4.3 Event Manager

The event manager is a wrapper around the event service of Orbacus. It is a Corba component serving as an entry point to the event service. It is responsible for creating event channels and maintaining event services.

4.4 Event Channels

Event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. It is both a supplier and consumer of events. Event Channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests.

4.5 Event Service Broker

The event service broker is a helper class for registering, deregistering, sending and receiving of events from the event service. Any of these commands can be issued in multiple forms by using overloaded methods. In the rest of the document, references to the event service broker and event manager are used interchangeably. The point to note here is that all clients deal with the event broker class only and the event broker class has a reference to the event manager. This saves the clients from explicitly resolving the event manager.

4.6 Type of Communication

A supplier can issue a single standard request to communicate event data to all consumers at once. Suppliers can generate events without knowing the identities of the consumers; conversely consumers can receive events without knowing the identities of the suppliers.

4.7 Structure of Events

The structure of any event sent using the Event manager is given below.

```
struct EventInfo
         {
               ChannelId channel;
               string EventId;
               DDate date;
               Any info;
         };
```

channel is a combination of type and name and uniquely identifies a communication channel. The field *info* carries the actual payload of any event.

5.0 Interfaces

5.1 Creating a channel

```
CosEventChannelAdmin::EventChannel_ptr createChannel(const char
         *type, const char *name);
```

Before creating a channel, that *type* of the channel has to be created. On getting a call to create a channel, the event service broker checks if that type already exists. If not it throws a *TypeNotFoundException*. Otherwise it returns a reference to the newly created channel or to an existing one with the same name and type.

5.2 Getting a channel

```
CosEventChannelAdmin::EventChannel_ptr  getChannel(const  char  *name,
         const char *type);

CosEventChannelAdmin::EventChannel_ptrgetChannel(const
         Gaia::Events::ChannelId &ch);
```

These functions are used for getting a reference to a channel already created. If the channel identified by the type and name is not found NULL is returned. Otherwise it returns the reference to the channel.

5.3 Destroying a channel

```
int destroyChannel(const char *type, const char *name);
```

This function is used to destroy an existing channel. No exception is thrown even if that channel does not exist.

5.3 Registering a Consumer

```
int registerConsumer(CosEventChannelAdmin::EventChannel_ptr evch,
                       CosEventComm::PushConsumer_ptr consumer);

int registerConsumer(const Gaia::Events::ChannelId &ch,
                       CosEventComm::PushConsumer_ptr consumer);

int registerConsumer(const char *name, const char *type,
                       CosEventComm::PushConsumer_ptr consumer);
```

These functions are used for registering a consumer with an event channel identified by channel id or name and type or channel pointer. On successful completion 0 is returned, else –1 is returned.

5.3 Sending an Event

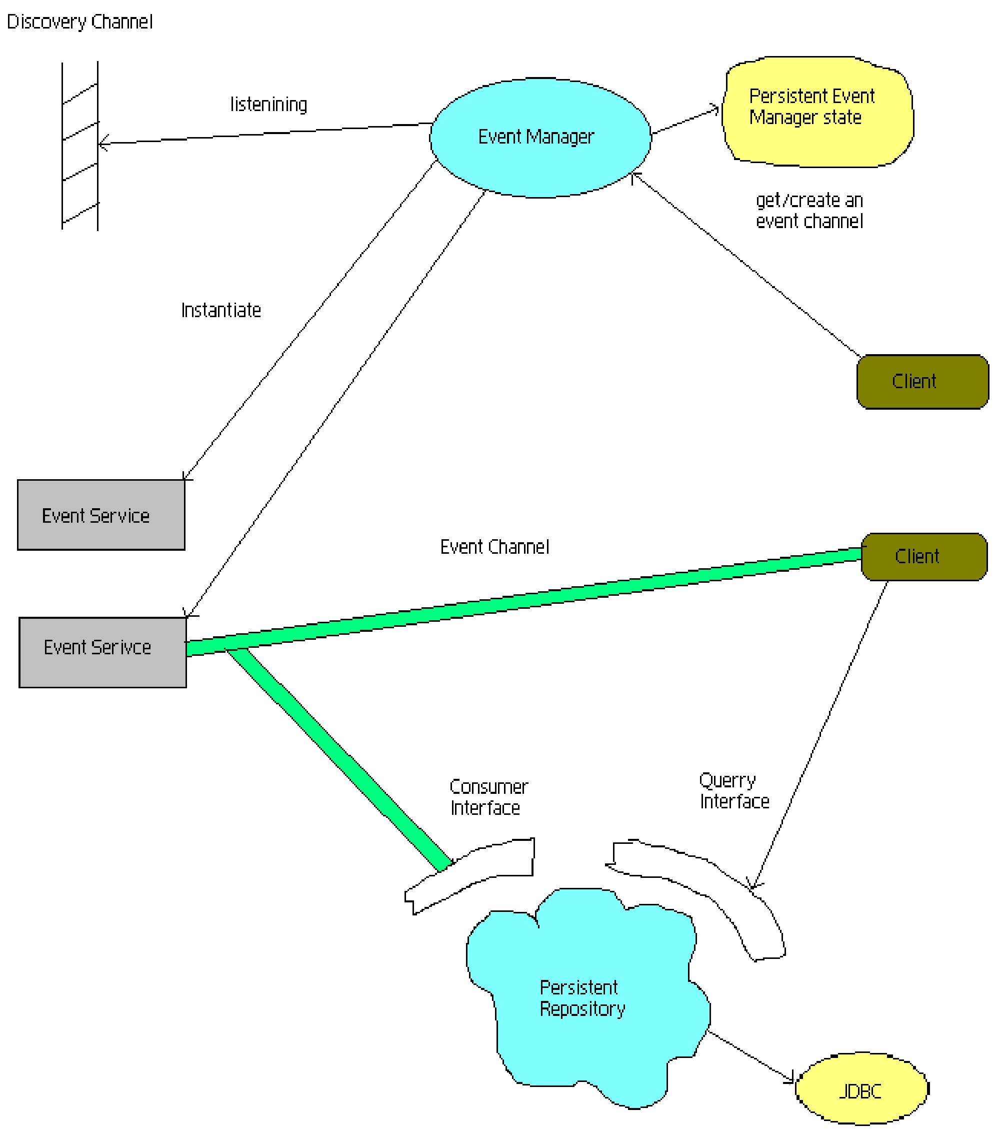
```
int send(CosEventChannelAdmin::ProxyPushConsumer_ptr channel, const
char *type, const char *name, const char * EventId, const CORBA::Any
& data);
```

This function is used to send an event on the channel identified by *channel*. Before dispatching the events to the consumers it updates the sending time and date in the event structure. On successful completion it returns 0, on failure it returns –1.

6.0 Proposed extensions

In a distributed system like an Active Space, it is highly possible that there are lots of entities which communicate through events. There is a discovery service which keeps track of the state of entities by sending events. In such a scenario, the number of events that an event service/channel has to handle may be enormous. This presents the event manager as the bottleneck in the entire system.

The issue of scalability can be handled by having a distributed event service system. What we are proposing here a collection of event services of CORBA and a single event manager coordinating and managing all these services. From the client's point of view, it sees only the event manager (event service broker) and event channels. The presence of numerous event services is transparent to it. This architecture enables easy integration of load distribution and persistent events. This scenario is illustrated in the following figure.



The current version of the event manager is efficient enough to handle latency requirements (refer to performance metrics of the event manager in appendix). The following paragraphs present a short description of the protocol used for achieving fault tolerance, load balancing, delayed query processing and persistence.

6.1 Protocol

6.1.1 Bootstrap

When the Active space bootstraps, an instance of the event manager is created and is registered with the naming service. Any client can communicate with the event manager after resolving it from the naming service. At the time of starting up, the event manager creates an event service in a remote machine. It stores the IOR of the event service it started in its lookup table (linked list) and goes into passive state waiting for clients to connect to it.

When a client wishes to create/ retrieve an event channel, or send / receive events, it connects to the Event Manager. The event system is organized such that the entry point to it is through the event manager. The clients are not aware of where the services or channel objects are actually located. They just get a reference to the channel object and communicate with it.

6.1.2 Creation

The consumer connects to the event manager passing it a name and type of the event channel it wishes to create. The event manager on getting the request looks up its internal table (described later in event manager state section) to see if such a channel is already present. If not, it runs an algorithm on its internal data structure, which keeps track of how many channel objects are present in any particular event service instance and decides whether a new event service is to be created, or a channel is created in one of the existing ones. The algorithm can check against a preset threshold, which defines the maximum number of channels (or the sum of the products of the number of channels and consumers in the channel) that an event service in the machine can handle without significant degradation of performance. It returns to the client the reference to the event channel (either a newly created on or an existing one).

6.1.3 Retrieval

When a consumer requests for any event channel, it looks up its internal tables to see if such a channel exists. If not it throws *ChannelNotFoundException*. If the channel exists it tries to connect to the event service to ensure that the event service is up and running. (This step is required because of fault tolerance – read the fault tolerance part for why this might be necessary). If the event service is up and running it returns the IOR of the channel to the client; else it throws a *ChannelNotFound* Exception.

6.2 Load Balancing

Typically the CORBA event service can handle approximately a thousand events per second (of 25 byte payload) when there is single producer and a single consumer. However as the number of consumers increase, the performance degrades drastically reaching three hundred events per second with 40 consumers. In a distributed event service system it should be possible to balance the load among various event services. The task of the event manager is essentially load balancing and keeping track of the event services. The Event manager keeps track of the number of event channels that is assigned to a particular event service and also the number of consumers in each channel. When a request for creating a new channel comes, using these two pieces of information it can take a decision as to whether or not it should instantiate a new service.

By load balancing we mean statically distributing different event channels among various instances of the event services. Once an event channel is created, there is no migration of that channel to another event service. Nor is there any splitting of event channels among different services.

6.3 Fault tolerance

The coordinator is a single point of failure in the system. There is no need for keeping track of the event services' state (alive or dead) as the discovery service keeps track of the state of all the entities in the system and each event service is an entity. If any of the event services crashes, the discovery service will send an event in the discovery channel and the event manager can listen to that channel to know whether any of the event service had gone down. It can then restart that service and update its internal data structures with the new IOR. It also has to update its internal data structure to reflect this change.

On the other hand, clients which were dealing with the event channels maintained by the event service that crashed, will get exceptions on trying to connect to the Event Channels. Then they can reconnect to the event manager to get the updated IOR of the new event channel. So every time a request to get or create an event channel comes to the event manager, it has to ascertain whether event service in which that particular event channel was created, is still alive and then return the IOR of the corresponding event channel.

6.4 Event Manager State

The event manager is stateful now as it keeps track of what event channels are available and how many consumers are subscribed to it. It maintains a table having the following data structure:

Event Service IOR	Name	Type	Event channel reference (list)	Number of Consumers

The event Service IOR is unique. The combination of name and type is also unique. The table is indexed on the basis of the name+type.

Since the event manager maintains state, there is a problem if the event manager crashes. For this it has to write the above internal data table to the file system so that it can read in those IOR when it is restarted. Once restarted it connects to each of the event services and makes a *getAllChannels* call to them. This is necessary to find out if any of the event services crashed during the absence of the event manager. Using this information it can rebuild its internal tables.

The state of the event manager can be tracked by the

1. Domain Service
   The problem of fault tolerance is common to many of the services in the Active Space like the File System or Discovery Service. So it seems better to have a separate service that keeps track of some essential services which are required for proper functioning of the entire system. There could three processes, which intermittently go to the name service and get the references of the *key* services and verify their status. If any of those go down this service can bring it up.

2. The event services in the system.

In the latter case, each of the ESs keep listening to the discovery channel for the state of the event manager. If any of them detect that the event manager has crashed, it can run a distributed election mechanism (refer to appendix for a possible election protocol) to elect one of the services as the coordinator. The service elected will instantiate a new event manager and the system will be restored. Since the state of the event manager is put in the common file system in a well-known directory, it is possible to start the new event manager in any machine and it still can access the previously saved state of the event manager that crashed.

6.3 Persistence

There are various services in Active Space, which require access to a persistent event heap. It is not very difficult to add persistence to this design. A persistent service will archive events for a certain period of time specified during creation of the event channel.

This persistent service can be our own implementation or it can be the event heap of Stanford. This persistent service should have the following features:

a) Query processing in terms of contents of the events
b) Storing events for time to live amount of time.

When the event manager starts up, it instantiates the event heap service. Not all events need to be persistent but that those need to be persistent has to be archived at the event heap for the designated amount of time. This decision can be made at the time of creation of a channel by specifying the type name beginning with p_ . If the event manager receives a request for creation of an event channel with the name beginning with p_ , it subscribes the event heap as a consumer of that channel. So any event generated in that channel will automatically and reliably be delivered to the event heap.

The event heap will internally maintain a database interacting with it through JDBC. So the query grammar will conform to the JDBC query grammar.

7.0 Appendix

7.1 Performance Metrics of the Event Service

These tests were carried out in a single Win2000 machine and there were no other applications running on it. Hence there was no network transfer associated with it. The payload in each event was **25 bytes**.

The supplier sent samples of 500, 1000 or 10000 events. The receive time below is measured as the difference between the time of sending the first event by the sender and receiving the last event in the receiver. So this takes care of latency of delivery of events.

The only parameters set for the event service in Orbacus were "concurrent thread pool" and Queue length (for buffering events). The following table shows the performance.

Number of events	Queue Length	Number of Consumers	Send time	Receive time	**Events/sec (send)**	**Events/sec (received)**
1000	5000	1	1	1	1000	1000
1000	5000	2	1	2	1000	500
1000	5000	4	2	3	500	333
500	5000	4	1	2	500	250
500	5000	8	2	4	250	140
500	5000	10	2	5	250	100
500	6000	12	2	6	250	84
500	9000	20	3	8	165	63
500	9000	40	6	17	84	29

7.2 Distributed Election Protocol

This protocol is for election of the event manager and it assumes that the other services in the Active Space (particularly the Discovery Service and the File System are running properly).

At the time of bootstrap, the event manager creates an *election* event channel and whenever it creates a new event service, it subscribes it to this *election* channel and sends an event on it specifying the number of event services currently active. Each of the event services keep listening to the discovery channel for the state of the event manager and when it detects that the event manager has crashed, it follows the following steps:

1. If it does not already get a proposal, it proposes itself as the coordinator in the *election* channel.
2. If at anytime after proposing itself as the coordinator, it gets a proposal from another event service having a higher IP address, then it backs off.
3. Every node that gets a proposal, has to reply to it with *yes* or *no* unless it has already proposed itself and its IP address is higher than the IP address of the sender of the new proposal.
4. When any proposer receives a quorum (maybe half of the number of event services votes for it) of affirmations for its proposal, it declares itself as the coordinator and announces it in the *election* channel. After a timeout period of time, it instantiates a new event manager.

This protocol is going to successfully terminate in instantiating a single event manager provided the events in the *election channel* are delivered reliably and within a *timeout* interval.

7.3 Sample code

All operations on the event manager can be done through the event service broker class. If repeated calls to the event manager is required, it is advised to have a class variable for ESBroker so that repeated resolving of the event manager is not necessary. The ESBroker class variable is instantiated as follows.

```
ESBroker esBroker;
if (esBroker.init(this) < 0)
      return -1;
```

All the examples below deal with the event channel of the following type.

```
char *type  =     “SYSTEM”;
char *name  =     “TEST”;
```

7.3.1 Creating a channel

```
      CosEventChannelAdmin::EventChannel_ptr echannel;
      try
      {
            echannel = esBroker.createChannel(type,name);
      }
      catch(Gaia::Events::EventManager::TypeNotFound)
      {
            // must create the type first else this exception
will be thrown.
      }
```

7.3.1 Retrieving a channel

```
CosEventChannelAdmin::EventChannel_ptr echannel;
echannel = esBroker.getChannel(type,name);
if(echannel == NULL )
      // printf (“Channel not found”);
```

7.3.3 Destroying a channel

```
if( esBroker.destroytChannel(type,name) = -1 )
      //printf (“error in deleting channel”);
```

7.3.4 Sending an Event

```
Gaia::Discovery::Discovered message;
CORBA::Any data;

data <<= message;

char * eventId = “testing”;
if ( esbroker.send( echannel.in(), "SYSTEM", "TEST",
eventId, data ) == -1)
      // printf(“error in sending data”);
```

7.3.5 Retrieving a Consumer

```
CosEventChannelAdmin::EventChannel_var echannel;

eChannel = esbroker.getConsumer("SYSTEM", "TEST");
if (CORBA::is_nil(eChannel.in()))
{
      OS::printToConsole("No consumer”);
      return -1;
}
```