International Journal of Information Technology and Business Management
29th December 2014. Vol.32 No.1

ISSN 2304-0777 www.jitbm.com

PUBLISH/SUBSCRIBE MECHANISM FOR PERVASIVE COMPUTING SYSTEMS

[1]Nlerum P. A. And [2]Nwachukwu E.O
[1]Department of Computer Science, Faculty of Science, Federal University Otuoka, Bayelsa State, Nigeria,
[2]Department of Computer Science, Faculty of Science, University of Port Harcourt, Rivers State, Nigeria
Email: [1]obukem@yahoo.com [2]enoch.nwachukwu@uniport.edu.ng

ABSTRACT

The publish/subscribe component model is an emerging paradigm to support software applications composed of highly evolvable and dynamic federations of components. According to this paradigm, components do not interact directly, but through a special-purpose element called EventService Broker. The Event Service receives events and forwards them to subscribers, that is, to all components registered to listen to them. The event paradigm and publish/subscribe systems allow clients to asynchronously receive information that matches their interests. We outline an event architecture for Pervasive Computing Systems that addresses two key requirements: component interaction and system-environment interaction. The system consists of access peer servers, event channels and a mechanism for locating events. The architecture uses filter covering and merging for supporting high accuracy in event delivery, reducing communication cost, and improving event processing on terminals and devices. Class and Block diagrams were used to implement the architecture.

Keywords: ***Event Service, Filter Covering, Publish/Subscribe***

1.0 Introduction

Pervasive computing system is an attempt to break away from the traditional distributed computing system to a more dynamic and ubiquitous computing paradigm. This evolving technology supports the utilization of computing infrastructures anywhere and anytime. Pervasive computing creates new possibilities for applications and services; however, it also presents new requirements for software that need to be taken into account in applications and in the service infrastructure. In order to support the development and deployment of intelligent applications a number of fundamental enabling middleware services are needed (Raatikainen, Christensen and Nakajima, 2002). Two important services are event monitoring and event notification, which are vital for supporting adaptation in applications. Event monitoring and notification are used to realize a number of pervasive applications, such as smart rooms, sensor networks, presence applications, and device tracking and management. Pervasive computing creates new requirements for event systems, such as mobility support, context-awareness, interoperability, and support for heterogeneous devices (Eugster, et al, 2003).

Publish/subscribe systems decouple the components participating in the communication: a sender does not know the receivers of its messages. Rather, receivers are identified by the dispatcher based on previous subscriptions. New components can dynamically join the federation, become immediately active, and cooperate with the other components without any kind of reconfiguration or refreshing of the modified application. They must simply notify the dispatcher that they exist by subscribing to particular events (Notkin et al., 1993).

Fig 1 below shows a model of a publish/subscribe architecture for pervasive components

International Journal of Information Technology and Business Management
29th December 2014. Vol.32 No.1

ISSN 2304-0777 www.jitbm.com

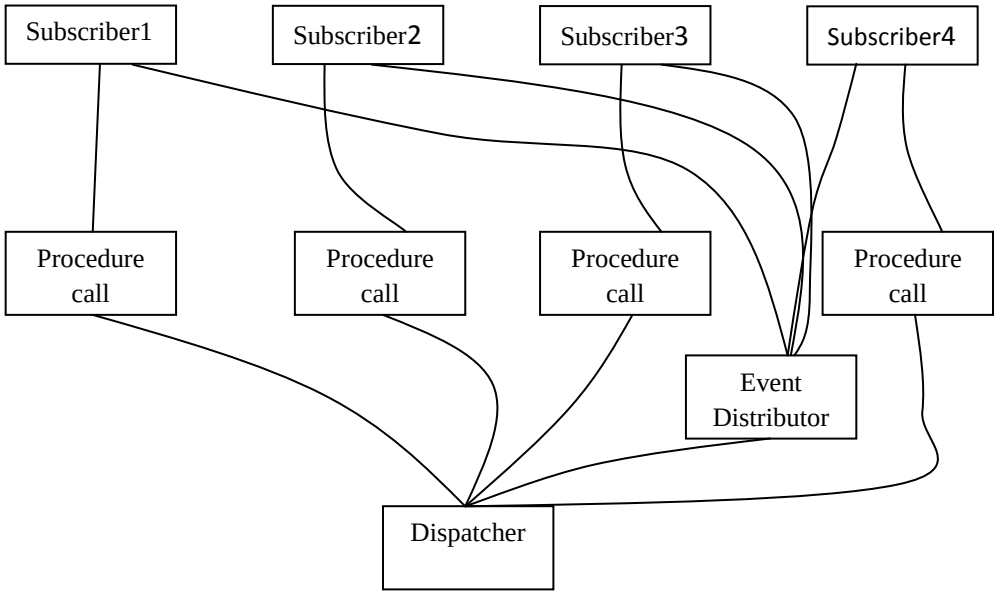


Fig. 1: Model of a Publish/Subscribe Architecture: (Eugster, et al, 2003).

The gain in flexibility is counterbalanced by the difficulty to the designer in understanding the overall behaviour of the application. Components are easy to reason about in isolation, but it is hard to get a picture of how they cooperate, and to understand their global behaviour. Although components might work properly when examined in isolation they can become faulty when put in a cooperative setting (Eugster, et al, 2003).

To solve these problems, the publish/subscribe architectures use a validation technique for component subscriptions and unsubscriptions. This problem arises at the architectural level, where each component specifies how it produces and consumes events. According to Garlan et al. 1992, the approach analyzes how the different elements cooperate to achieve a common goal, but does not address how they behave in isolation. Individual components are assumed to work as specified.

There is wide consensus that analysis and testing are complementary rather than competing techniques for validating software systems and improving their quality (Young and Taylor, 1989). This contribution stresses validation by proposing an innovative technique to analyse publish/subscribe systems that helps discover problems early during the design phase, but that does not replace component, integration, and system tests. Publish/subscribe systems are often implemented on top of a middleware platform that provides all event dispatching features. Implementors have only to configure it. The other components must be designed and coded explicitly. Modeling and validation exploits this peculiarity. Modeling the EventService is a complex task, but the effort is counterbalanced by the fact that the dispatcher is usually reused in different systems. The accurate representation of such a component would clearly impact the whole design phase, but it's being application-independent eases the task. By following a reuse approach at the modeling stage, a parametric model of the dispatcher is provided. Developers have only to configure it to render the communication paradigms they want to use. In contrast, application-specific components must be specified as UML state-chart diagrams (Prieto-Diaz and Neighbors, 1996). Once all components have been designed in isolation, modeled systems are validated through model checking.

International Journal of Information Technology and Business Management
29th December 2014. Vol.32 No.1

ISSN 2304-0777 www.jitbm.com

2. Modeling Events

When modeling events, one must think about how to decompose communication in a system into generic messages that convey certain standard information in the form of an event object. Such an object could be of any type, but there are certain advantages to requiring event types to be subclasses of an Event base class as shown in Figure 2. One advantage is that the Event class can enforce a common interface or set of attributes on all events, such as a time stamp which captures the time of occurrence of an event. Such an event hierarchy also enables type-based subscription: subscribers can specify a non-leaf node in the event tree to be informed of all events of that type or below.

The natural way to implement an event hierarchy in an object-oriented language is by using inheritance. However, it could also be implemented using a string representation, with the levels of the hierarchy separated by colons. For example, the event types in Figure 2 might be represented as:

Event, Event:ManagementEvent,
Event:ManagementEvent:FaultEvent,

and so forth.

The inheritance mechanism, not surprisingly, has several advantages. Principally, it allows an event subscriber to more easily subscribe to all events in a subtree of the type hierarchy. One subscriber

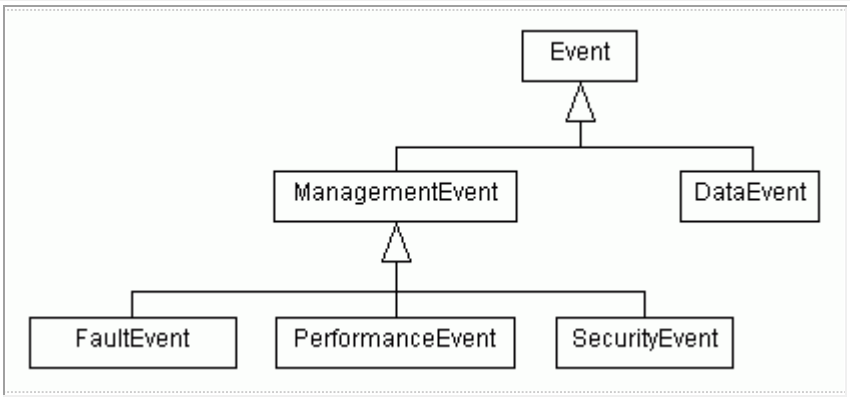


Fig. 2: Example Event Hierarchy

may want all management events, while another may only be interested in data events. This is expressed simply by subscribing to the event at the root of the hierarchy of interest. Furthermore, since events are objects, attributes may be packaged naturally with them, so the subscriber need not invoke methods on (say) the publisher to retrieve those attributes. The inheritance mechanism also takes advantage of compile-time type checking. The compiler will not catch errors in string representations, such as a misspelled event type, but it will catch errors in class names. The advantages to the string technique are that it works in languages that are not object-oriented, and strings may be generated dynamically, while in many languages, objects in an event hierarchy must have classes that are defined at compile time (Suchitra Gupta, Jeff Hartkopf and Suresh Ramaswamy, 2002)

3. System Architecture

Figure 3 below shows a high level view of our architecture. In a system of any size, we would like to minimize the number of dependencies and interconnections between objects to keep the system from becoming brittle and hard to change. The more dependencies there are, the more a change in any particular point in the system propagates to other points in the system. The simplistic approach requires each managed object to maintain a reference to each management system. The number of these references increases geometrically with the number of managed objects and management systems. A better approach

International Journal of Information Technology and Business Management
29th December 2014. Vol.32 No.1

ISSN 2304-0777 www.jitbm.com

that keeps this to a linear increase is to have a mediator that encapsulates and coordinates the communication. The event service offers this service as shown below.

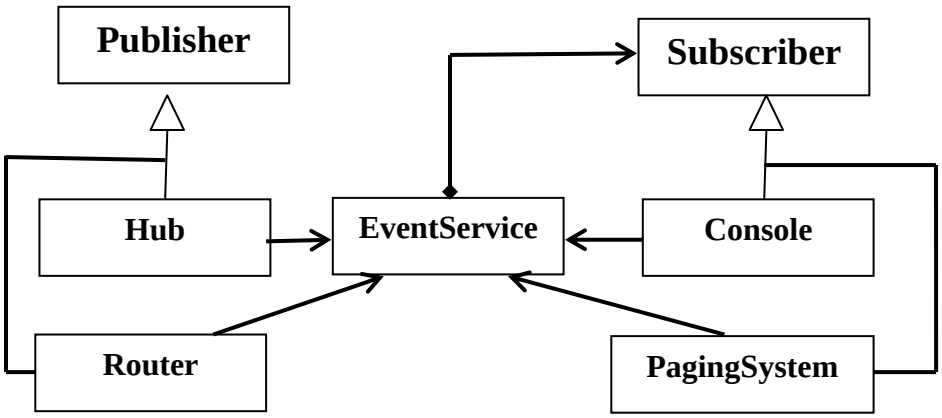


Figure 3: Architecture of a Publish Subscribe Mechanism.

Event Notifier derives many of its benefits from the fact that subscribers only know about events, not about publishers. For example, routers and hubs might both generate events of the same type, say FaultEvent, when problems occur. In an Observer implementation, each management system needs to know which managed objects generate fault events and register with each. The same is essentially true of Mediator, except that the mediator encapsulates this knowledge. However, using Event Notifier, a management system needs only to register interest in the FaultEvent type to get all fault events, regardless of who publishes them.

4. Architecture Implementation

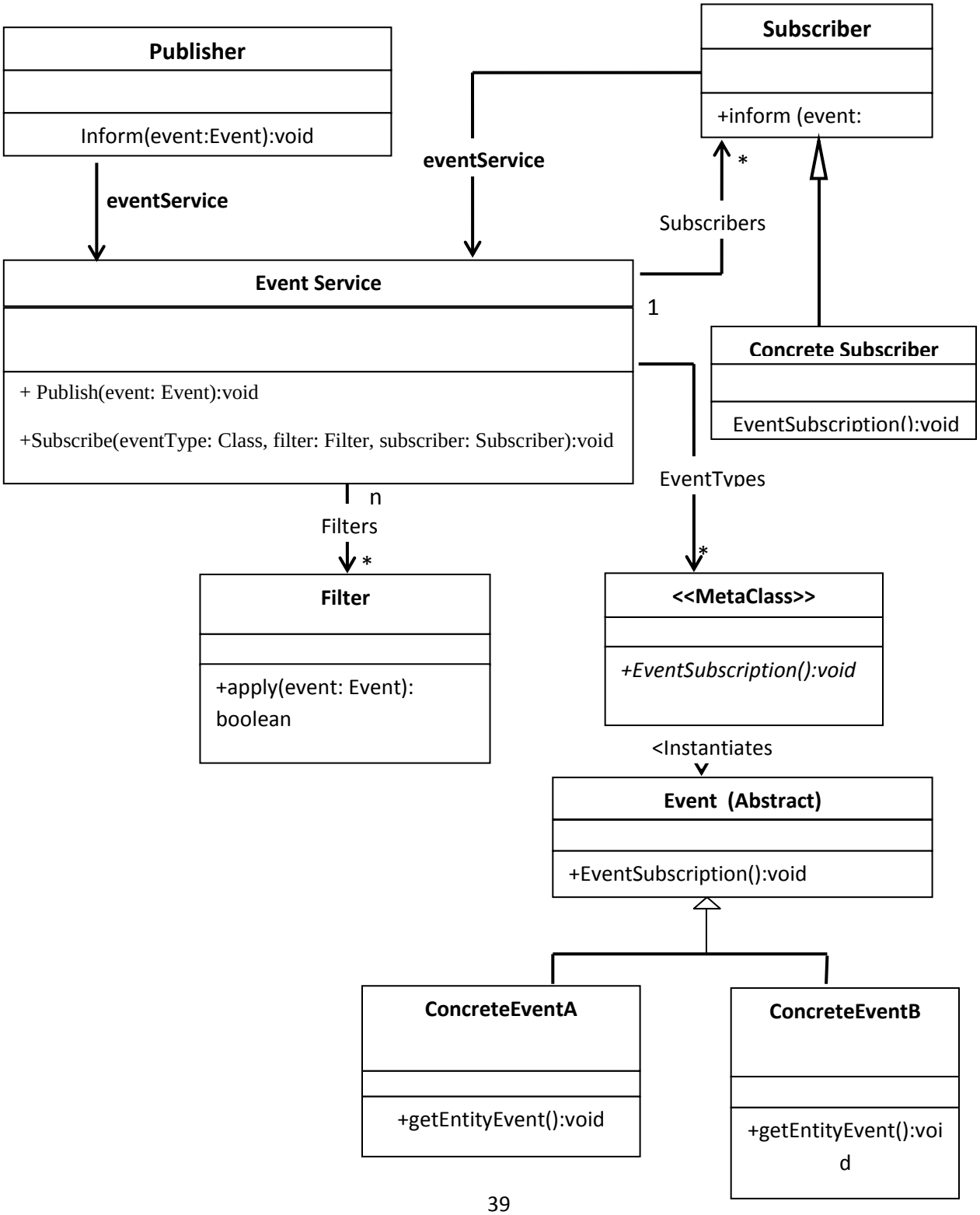
In this section, we implement the publish/subscribe architecture. Figure 4 shows the class diagram of our implemented architecture. Figure 5 shows collaboration diagram for the implemented architecture. It displays the logical interaction among major components of the architecture.

For each of the components , the subscriber invokes the subscribe method on the event service specifying the event type it is interested in and passes a reference to itself (or possibly another object) and a filter. The eventType argument represents the type of the event. When an event occurs, the publisher invokes the publish method on the event service passing an event object. The event service determines which subscribers are interested in this event, and for each of them applies the filter provided by the subscriber. If no filter is provided, or if application of the filter results in true, then the event service invokes the inform method of the subscriber, passing the event as an argument.

All publication and subscription is done through the event service. The event service maintains all information about which subscribers are interested in which events, so that publishers and subscribers need not be aware of each other. Moreover, anyone may publish or subscribe to events using the well-defined interface provided by the event service without having to implement any special logic to handle interaction with other entities for which it has no other reason to communicate.

International Journal of Information Technology and Business Management
29th December 2014. Vol.32 No.1

ISSN 2304-0777 www.jitbm.com



ISSN 2304-0777 www.jitbm.com

Fig 4: Event Notification Class Diagram

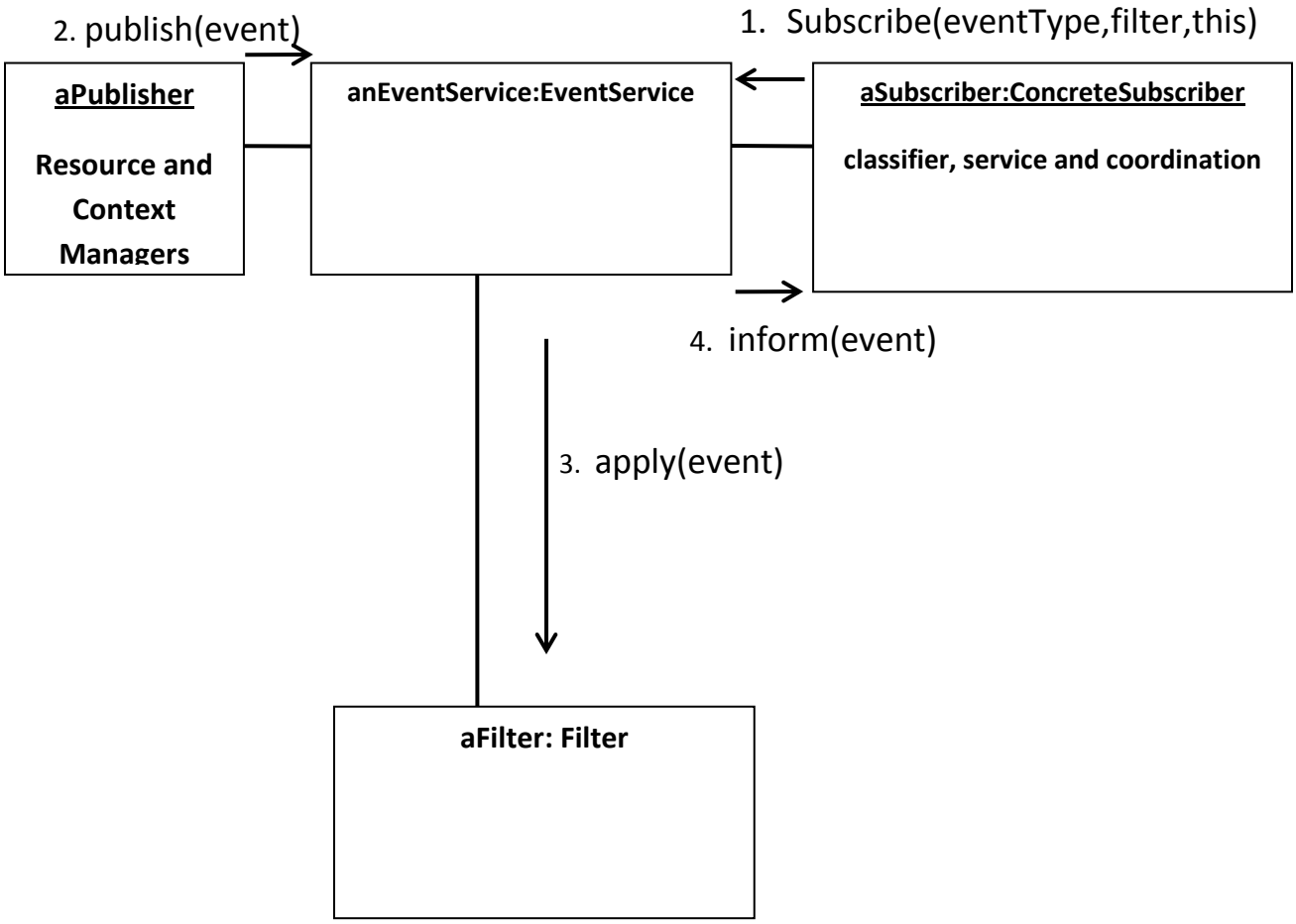


Fig. 5: Event Notifier Collaboration Diagram

5. Table Summary of EventNotifier Publish/Subscribe Mechanisms for Mediator and Observer Types.

This section compares and contrasts other closely related patterns with Event Notifier. Event Notifier is closely related to the Observer and Mediator pattern (Gamma,et al, 1995), and in fact, can be viewed as a cross between them. Table 1 summarizes similarities and differences among the patterns.

International Journal of Information Technology and Business Management
29th December 2014. Vol.32 No.1

ISSN 2304-0777 www.jitbm.com

Table 1. Comparison between Mediator and Observer Patterns

	Mediator	Observer	Event Notifier
Knowledge of who can publish and whom to notify of events	Localized in the mediator	Duplicated by each observer and subject	Localized in the event service
Coordination of behavior in response to events	Managed exclusively by the mediator	Managed by participating observers	Managed by participating subscribers
Messaging flexibility	Application-specific messages through methods defined at compile time	Single generic message through the predefined update() method	Flexible messaging by using different event types
Informative content of messages	Methods may be defined to contain information as desired	Predefined update() method has no arguments, so no information is carried in the message	Events may be defined to contain information as desired
Registration of interest in notification	Interest in notification is embedded in the mediator	Observers may register interest dynamically	Subscribers may register interest dynamically
Coupling between publishers and subscribers	Mediator decouples colleagues	Observers and subjects are coupled	Event service decouples publishers and subscribers
Relationship between publishers and subscribers	Many-to-many	One-to-many, as an observer may only observe a single subject	Many-to-many

6. Event Service Sample Code

This section provides a skeletal implementation of Event Notifier Publish/Subscribe scheme in Java, and shows it can be used.

Event Service:

The EventService class is implemented as a singleton, providing a well-known point of access and implementations for publish, subscribe, and unsubscribe.

```
public class EventService {
  // Prevents direct instantiation of the event service
  private EventService() {
    eventClass = Event.class;
    subscriptions = new Vector();
  }

  static private EventService singleton = null;

  // Provides well-known access point to singleton EventService
  static public EventService instance() {
    if (singleton == null)
      singleton = new EventService();
    return singleton;
  }

  public void publish(Event event) {
    for (Enumeration elems = subscriptions.elements(); elems.hasMoreElements(); ) {
      Subscription subscription = (Subscription)
```

International Journal of Information Technology and Business Management
29th December 2014. Vol.32 No.1

ISSN 2304-0777 www.jitbm.com

```
elems.nextElement();
    if
(subscription.eventType.isAssignableFrom(event.get
Class())
       && (subscription.filter == null ||
subscription.filter.apply(event)))
      subscription.subscriber.inform(event);
   }
  }

  public void subscribe(Class eventType, Filter filter,
Subscriber subscriber)
    throws InvalidEventTypeException {
     if (!eventClass.isAssignableFrom(eventType))
       throw new InvalidEventTypeException();

     // Prevent duplicate subscriptions
     Subscription subscription = new
Subscription(eventType, filter, subscriber);
     if (!subscriptions.contains(subscription))
       subscriptions.addElement(subscription);
  }

  public void unsubscribe(Class eventType, Filter
filter, Subscriber subscriber)
    throws InvalidEventTypeException {
     if (!eventClass.isAssignableFrom(eventType))
       throw new InvalidEventTypeException();
     subscriptions.removeElement(new
Subscription(eventType, filter, subscriber));
  }

  private Class eventClass;
  protected Vector subscriptions;
}

// Stores information about a single subscription
class Subscription {
  public Subscription(Class anEventType, Filter
aFilter, Subscriber aSubscriber) {
    eventType = anEventType;
    filter = aFilter;
    subscriber = aSubscriber;
  }

  public Class eventType;
  public Filter filter;
  public Subscriber subscriber;
}

public class InvalidEventTypeException extends
RuntimeException {}
```

Subscriber

An abstract subscriber class and an example concrete subscriber are shown below. The PagingSystem subscriber subscribes to and handles FaultEvent.

```
public abstract class Subscriber {
  abstract void inform(Event event);
}

public class PagingSystem extends Subscriber {
  public PagingSystem() {
    FaultEvent event = new FaultEvent();
    CriticalFaultFilter filter = new
CriticalFaultFilter();
    EventService.instance().subscribe(filter.getClass(),
filter, this);
  }

  public void inform(Event event) {
    // Assumes that this subscriber has only subscribed
to FaultEvent
    FaultEvent faultEvent = (FaultEvent) event;

    // Respond to the event
    System.out.println("Critical Fault Event
occurred:" + faultEvent);
  }
}
```

Publisher

The Router class is an example publisher, which publishes a critical fault event.

```
public class Router {
  public static void triggerPublication() {
    Event event = new
FaultEvent(FaultEvent.CRITICAL);
    EventService.instance().publish(event);
  }
}
```

ISSN 2304-0777 www.jitbm.com

7. Discussions

This section discusses specific issues that one must address when implementing eventnotifier Publish/Subscribe architectures in a real pervasive computing situation. We divide the issues into more or less autonomous sections, discussing implementation techniques where appropriate.

Large scale use of Event Notifier can potentially result in the event service being a single point of failure and a performance bottleneck. One solution to this problem is to have multiple event services, where each brokers a group of related events; for example, one event service for network management, and another for financial services. The failure of the event service for one area will not impact the systems dependent only on events from the other area.

Another way to deal with the problem is to use a variation of Event Notifier in which the subscription list is not stored in the event service, but is instead distributed across publishers. One can achieve this without breaking the semantics of a centralized event service by having publishers advertise the event types they intend to publish. The event service would maintain this association of publishers and event types. On subscription to an event type, the event service would pass the subscriber to the publishers of that event type. Now publishers can inform subscribers directly.

Since the subscription list is distributed among the set of publishers instead of being stored centrally in the event service, failure of any single component is likely to have less impact on the system as a whole. Since the frequency of publication is typically much higher than the frequency of subscription, and at publication time events are not being channeled through a single point, there is no single point that could become a bottleneck.

Since the publication list changes less frequently than the subscription list, it is feasible to maintain a "hot spare" for the event service. Whenever an advertisement occurs, the event service updates the hot spare, thus keeping the information current in both places. If the event service goes down, the hot spare can take over.

8. Conclusion/Future Work

A prototype implementation of the proposed system has been developed using the Java language, building on existing technologies such as SOAP and Apache Axis. We have also developed a lightweight version of the client-side API, which features pub/sub and session management operations for J2ME MIDP (Mühl, 2010). The present application uses the event system to disseminate changes in users' presence information. Currently, load balancing and federation support are under development. The framework supports scalability to a number of event servers, but wide-area scalability is an open issue. Dynamic filtering and merging in both client-server and server-server environments seem to be promising research topics—how to balance between the precision and size of filter sets using different algorithms. These mechanisms may also be used in ad hoc environments on devices that have enough processing power. The proposed event architecture is envisaged to be a supporting layer for a compound event detection system that supports the detection of complex event sequences in time.

Summarily the Event Notifier pattern should be used in any of the following situations:

- When an object should be able to notify other objects of an event without needing to know who these objects are or what they do in response.
- When an object needs to be notified of an event, but does not need to know where the event originated.
- When more than one object can generate the same kind of event.
- When some objects are interested in a broader classification of events, while others are interested in a narrower classification.
- When an object may be interested in more than one kind of event.
- When you need to dynamically introduce new kinds of events.

ISSN 2304-0777 www.jitbm.com

- When objects participating in notification may be dynamically introduced or removed, as in distributed systems.
- When you need to filter out events based on arbitrary criteria.

References

1. Caporuscio, M., Inverardi, P., Pelliccione, P. (2002), *Formal Analysis of Clients Mobility in the Siena Publish/Subscribe Middleware*, Technical Report, Department of Computer Science, University of Colorado.
2. Eugster, P. T., Felber, P. A., Guerraoui, R. and Kermarrec, A. (2003), *The many faces of publish/subscribe.* ACM Computing Surveys, 35(2): pages114–131.
3. Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
4. Garlan, G., Shaw, M, Okasaki, C., Scott, C. and Swonger, R. (1992), *Experience with a course on architectures for software systems*. Iin Proceedings of the Sixth SEI Conference on Software Engineering Education, Springer-Verlag, LNCS Pages 376-393.
5. Mühl, G. (2010), *Generic Constraints for Content-based Publish/Subscribe*, in: C. Batini et al. (Eds.), CooPIS 2010, LNCS 2172, pp. 211-225, Springer-Verlag, 2010.
6. Notkin, D., Garlan, D., Griswold, W. G. & Sullivan, K. (1993), *Adding Implicit Invocation to Languages: Three Approaches.* LNCS 742, ISOTAS, Conference Proceedings, pages 489-510.
7. Prieto-Diaz, R. and Neighbors, J. M. (1996), *Module Interconnection Languages*. Journal of Systems and Software 6 (4), 307-334.
8. Raatikainen, K., Christensen, H., Nakajima, t. (2002), *Application Requirements for Middleware for Mobile and Pervasive Systems*, ACM SIGMOBILE Mobile Computing and Communications Review, Volume 6, Issue 4.
9. Young, M. and Taylor, R. N. (1989), *Rethinking theTtaxonomy of Fault Detection Techniques.* In Proceedings of the 11th International Conference on Software Engineering, pages 53–62.