The Many Faces of Publish/Subscribe

P.Th. Eugster,[1] P. Felber,[2] R. Guerraoui,[1] and A.-M. Kermarrec[3]

[1] Swiss Federal Institute of Technology, Lausanne, CH
[2] Bell Laboratories, Murray Hill, USA
[3] Microsoft Research Ltd., Cambridge, UK

Abstract

Well-adapted to the loosely coupled nature of distributed interaction in large scale applications over the Internet, the publish/subscribe communication paradigm has recently received an increasing attention. With systems based on that paradigm, subscribers register their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events fired by publishers. Many variants of the paradigm have recently been promoted, each variant being specifically adapted to some given application or network model. This paper factors out the common denominator underlying these variants: full decoupling of the communicating participants in time, space and flow. We use these decoupling dimensions to better identify commonalities and divergences with traditional interaction paradigms. The many existing variations on the theme of publish/subscribe are classified and synthesized. In particular, their respective benefits and shortcomings are discussed both in terms of interfaces and implementations.

1 Introduction

The Internet has considerably changed the scale of distributed systems. Distributed systems now involve thousands of entities — potentially distributed all over the world — whose location and behavior may vary throughout the lifetime of the system. These constraints visualize the demand for more flexible communication models and systems, reflecting the dynamic and decoupled nature of the applications. Individual *point-to-point* and *synchronous* communications lead to rigid and static applications, and make the development of dynamic large scale applications cumbersome. To reduce the burden of application designers, the glue between the different entities in such large scale settings should rather be provided by a dedicated communication engine, based on an adequate communication scheme.

The *publish/subscribe* interaction scheme is now receiving increasing attention and is claimed to provide the loosely coupled form of interaction required in such large scale settings. Subscribers have the ability to express their interest in an event, or a pattern of events, and they are notified afterwards of any event fired by a publisher, matching their registered interest. An event is asynchronously notified to all subscribers that registered interest in that given event. The strength of this event-based interaction style relies on a full decoupling in *time, space* and *flow* between publishers and subscribers. Many industrial systems and research prototypes support this style of interaction, and there is a proliferation of research prototypes on novel forms of publish/subscribe interaction schemes. However, because of the multiplicity of these systems and prototypes, it is rather difficult to capture their commonalities, and draw a sharp line between their main variations.

The aim of this paper is threefold. First we point out the common denominators of publish/subscribe schemes: *time, space* and *flow* decoupling of subscribers and publishers. These decoupling dimensions are illustrated by comparing the publish/subscribe paradigm with "traditional" interaction schemes. Second, we compare the many variants of publish/subscribe schemes: namely, *topic-based, content-based* and *type-based*. Third, we discuss variations and tradeoffs in the design and implementation of publish/subscribe-based systems through specific examples.

The paper focuses on interaction abstractions behind specific implementation details and communication infrastructures. Typical distributed system infrastructures usually provide several interaction model implementations. We shall indeed use specific infrastructures to illustrate some interaction issues and solutions but the aim is not to survey these infrastructures.

2 The Basic Interaction Scheme

The publish/subscribe interaction paradigm provides subscribers with the ability to express their interest in an event or a pattern of events, in order to be notified afterwards of any event fired by a publisher, matching their registered interest. In other terms, producers publish information on a software bus (an event manager) and consumers subscribe to the information they want to receive from that bus.

The basic system model (Figure 1) for publish/subscribe interaction relies thus on an event notification service providing storage and management for subscriptions as well as efficient delivery of notifications. Such an event service represents a neutral mediator between publishers, acting as producers of notifications, and subscribers, acting as consumers of notifications. Subscribers register their interest in events by typically calling a `subscribe()` operation on the event service, without knowing the effective sources of these events. This subscription information remains stored in the event service and is not forwarded to publishers. The symmetric operation `unsubscribe()` terminates a subscription.

To fire an event, a publisher typically calls a `notify()` operation. The event service directs the call to all relevant subscribers, and can in that sense be viewed as an untyped proxy for the subscribers. Note that *every* subscriber will receive a notification for *every* event conforming to its interest (obviously, failures might prevent subscribers from receiving notifications). Publishers typically use `publish()` and `register()` operations and may also have the ability to advertise the nature of their future notifications through an `advertise()` operation (e.g., Siena or CEA — see Sidebar) The provided information can be useful for (1) the event service to adjust itself to the expected flows of events, and (2) the subscribers to learn about new available information.

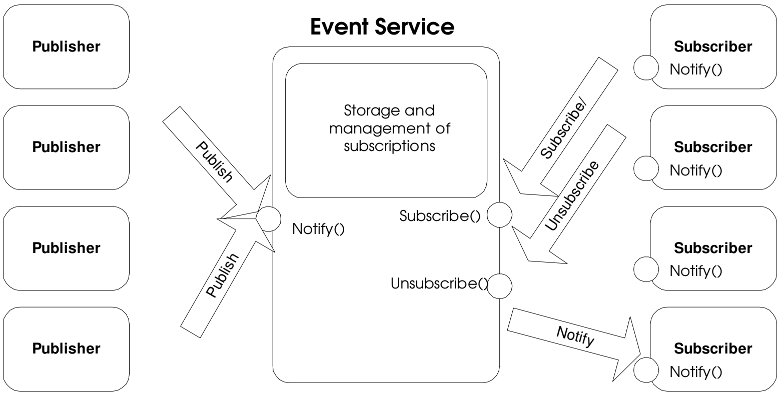


Figure 1: A simple object-based publish/subscribe system

The decoupling that the event service provides between publishers and subscribers can be decomposed along the three following three dimensions (Figure 2).

- **Space decoupling**: the interacting parties do not need to know each other. The publishers publish events through an event service and the subscribers get these events indirectly from the event service. The publishers do not usually have any reference to these subscribers, neither do they know how many of these subscribers are participating in the interaction. Similarly, subscribers do usually not have any reference to the publishers, neither do they know how many of these publishers are participating in the interaction.

- **Time decoupling**: the interacting parties do not need to be actively participating in the interaction at the same time. In particular, the publisher might publish some events while the subscriber is disconnected, and conversely, the subscriber might get notified about the occurrence of some event while the original publisher of the event is disconnected.

- **Flow decoupling**: publishers are not blocked while producing events, and subscribers can get notified about the occurrence of some event while performing some concurrent activity (through a callback), i.e., subscribers do not need to pull for events in a synchronous manner. In short, message production and consumption do not happen in the main flow of control of the publisher or subscriber.

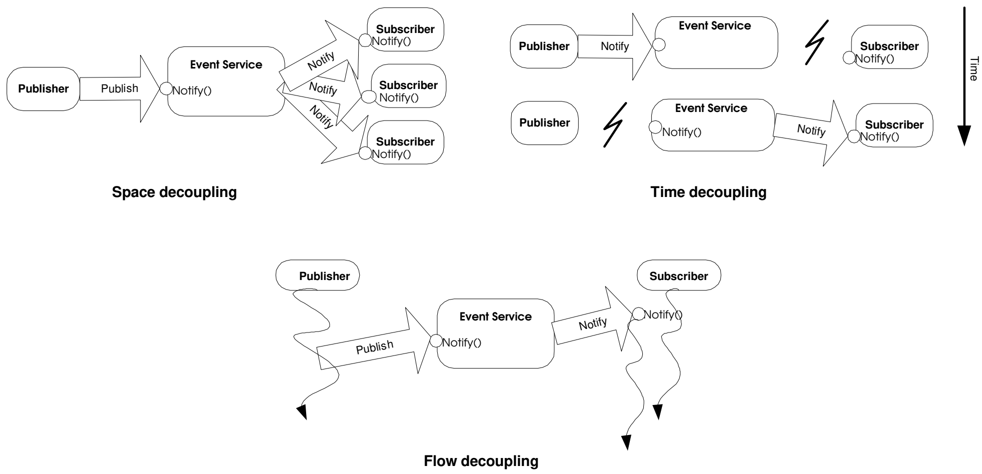


Figure 2: Space, time and flow decoupling with the publish/subscribe paradigm

Decoupling, in terms of time, space and flow of the production and consumption of information increases scalability by removing all explicit dependencies between the interacting participants. In fact, removing these dependencies strongly reduces coordination and thus synchronization between the different entities.

3 The Cousins: Alternative Communication Paradigms

Message passing, *remote invocations*, *notifications*, *shared spaces* and *message queuing* do all constitute alternative communication paradigms to the publish/subscribe scheme. We emphasize here their inability to provide full decoupling between participants.

Message passing

Message passing can be viewed as the ancestor of distributed interaction. Tightly coupled to the *socket* abstraction, message passing represents a very low-level form of distributed communication.

More complex interaction schemes are build on top of sockets still, but message passing is nowadays rarely directly used for developing distributed applications, since physical addressing and data marshaling, and sometimes even flow control (e.g., retransmission), become visible to the upper application layer. Message passing is asynchronous for the producer, while message consumption is generally synchronous. The producer and the consumer are coupled both in time and space (cf. Figure 3): they must both be active at the same time and the destination of a message is explicitly specified: the producer knows the consumer.

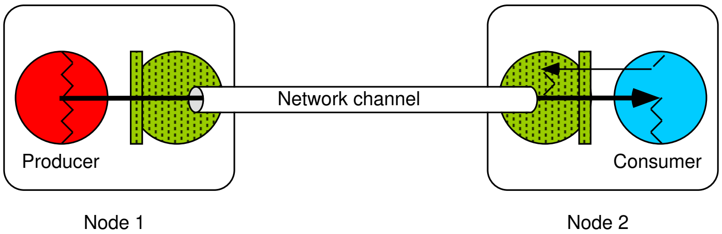


Figure 3: Message passing interaction — The producer sends messages asynchronously through a communication channel (previously set up for that purpose). The consumer receives messages by listening synchronously on that channel.

RPC

One of the most known forms of distributed interaction is based on extending the notion of "operation invocation" to a distributed context, i.e., by providing a form of remote invocation between distributed participants. This type of interaction has first been proposed in the form of remote procedure call (RPC — see Sidebar) for procedural languages, and has been straightforwardly applied to object-oriented contexts in the form of remote method invocations, e.g., Java RMI, CORBA, Microsoft DCOM (see Sidebar for more details on these). This multiplicity is responsible for diverging terminologies, e.g., "request/reply", "client/server".

By making remote interactions appear the same way as local ones, the RPC and its derivatives make distributed programming very easy. This explains their tremendous popularity in distributed computing. Remote interactions are however by their very nature different from local ones, e.g., by giving rise to further types of potential failures. This difference becomes more important at an increasing scale, especially because the inherent *one-to-one* semantics cannot be efficiently used to disseminate information to several interested parties. As shown in Figure 4, RPC is opposite to publish/subscribe in terms of coupling: the synchronous nature of RPC introduces a strong time, flow (on the consumer side[1]), and also space coupling (since an invoking object requires a remote reference to each of its invokees).

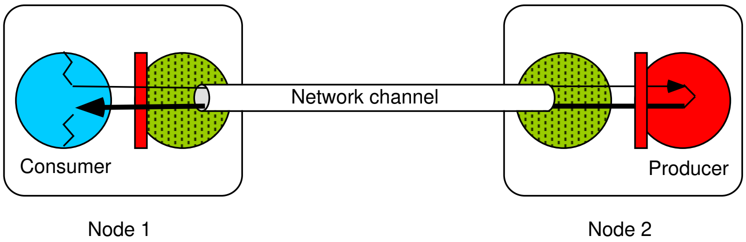


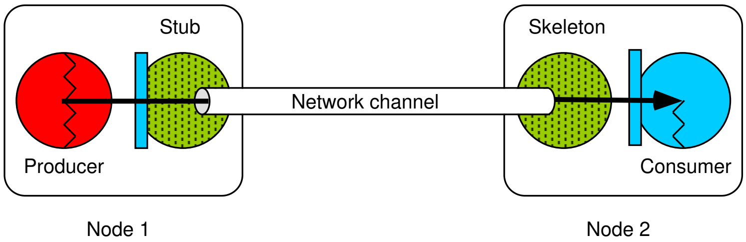
Figure 4: RPC and derivatives — The producer performs a synchronous call, which is processed asynchronously by the consumer.

Several attempts have been made to reduce coupling in remote invocations, especially targeting at removing flow coupling in order to avoid blocking the caller thread on the reply of a remote invocation. A first variant consists in providing some form of asynchronous invocation for remote operations without return values (e.g., CORBA *oneway*), as shown in Figure 5(a). This leads to invocations with weak reliability guarantees because the sender does not receive success or failure notifications (*fire-and-forget*). The second, less restrictive type, supports return values of remote invocations. However, instead of making an invoking thread wait for a result, a handle is directly returned as return value of a request. With this variant, known as *future* or *future type message passing* (cf. Sidebar), the invoking thread can immediately proceed and request the return value later, thanks to the handle (Figure 5(b)).

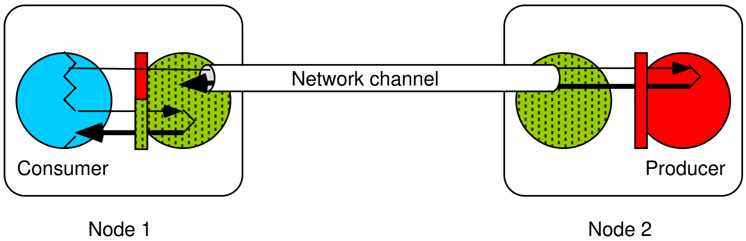
Notifications

In order to achieve flow decoupling, a synchronous remote invocation is sometimes split into two asynchronous invocations: the first one sent by the client to the server — accompanied by the invocation arguments and a callback reference to the client — and the second one sent by the server to the client to return the reply. This scheme can be easily used to express several replies by having the server make several callbacks to the client. Such notification-based interaction is for instance widely used to ensure consistency of Web caches. This implements a limited form of publish/subscribe interaction in which Web proxies act as subscribers and the Web sever as the

[1]The distinction of consumer/producer roles is not straightforward in RPC. We assume here that an RPC which yields a reply attributes a consumer role to the invoker, while the invokee acts as producer. As we will point out, the roles are inversed with *asynchronous* invocations (no reply).



(a) Asynchronous invocation — The producer does not expect a reply.



(b) Future invocation — The producer is not blocked and can access the reply later when it becomes available.

Figure 5: Decoupling flows with remote invocations

publisher.

This type of interaction — where subscribers register their interest directly with publishers, which manage subscriptions and send notifications — corresponds to the so-called *observer design pattern* (see Sidebar), illustrated by Figure 6. This communication style is often build on asynchronous invocations in order to enforce flow decoupling and corresponds thus more to a scheme than to a class of abstractions or mechanisms. Though publishers asynchronously notify subscribers, both remain coupled in time and in space. Furthermore the communication management is left to the publisher and can become very burdensome as the system grows in size.

Shared spaces

The *distributed shared memory* (DSM) paradigm provides hosts in a distributed system with the view of a common shared space across disjoint address spaces, in which synchronization and communication between participants take place through operations on shared data. The *tuple space* [2].

[2]This paradigm has been originally integrated at the language level in Linda [5]

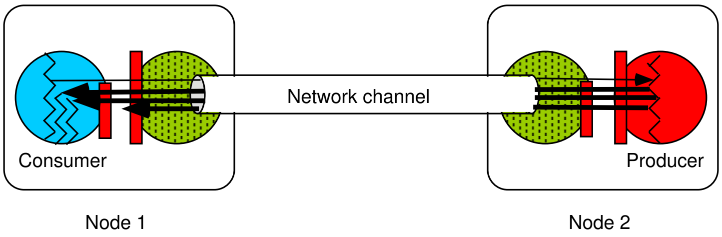


Figure 6: Notifications — Producers and consumers communicate using asynchronous invocations flowing in both directions.

provides a simple and powerful shared memory abstraction. A tuple space is composed of a collection of ordered tuples, equally accessible to all hosts of a distributed system. Communication between hosts takes place through the insertion and removal of tuples to/from the tuple space. Three main operations can be performed: `out()` to export a tuple in a tuple space, `in()` to import a tuple from the tuple space [3] and `read()` to read without withdrawing a tuple from the tuple space.

The interaction model provides time and space decoupling in that tuple producers and consumers remain anonymous with respect to each other. The creator of a tuple needs no knowledge about the future use of that tuple or its destination. An `in`-based interaction implements *one-of-n* semantics (only one consumer reads a given tuple) whereas `read`-based interaction would be able to provide the *one-to-n* message delivery (a given tuple can be read by all consumers). However, there is no flow decoupling as in the publish/subscribe paradigm since consumers pull new tuples from the space in a synchronous style (Figure 7). This limits the scalability of the model due to the required synchronization between the participants. [4]

Message queuing

Message queuing is a more recent alternative for distributed interaction. In fact, the term message queuing is often used to refer to a family of products rather than to a specific interaction scheme. Message queuing and publish/subscribe are tightly intertwined: message queuing systems usually integrate some form of publish/subscribe-like interaction. Such message-centric approaches are often referred to as *message-oriented middleware* (MOM — see Sidebar).

[3]Note that after a `in()` operation, the tuple does no longer belong to the tuple space.

[4]To compensate the lack of flow decoupling, new primitives such as asynchronous notifications have been integrated into the design of more recent solutions, as for instance in JavaSpaces.JavaSpaces extend the Linda tuple space model with asynchronous notifications through a corresponding `notify()` operation to register a subscriber.

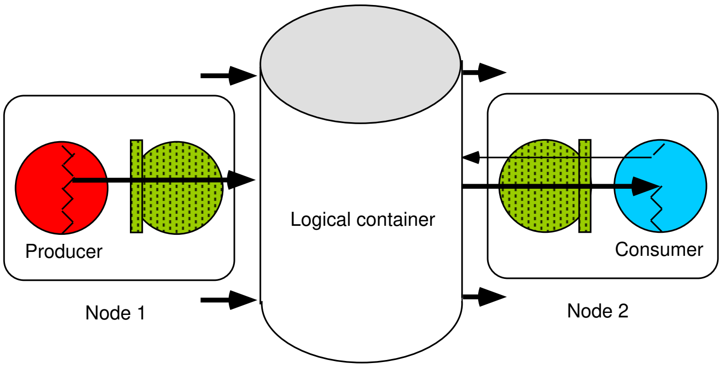


Figure 7: Shared space — Producers insert data asynchronously into the shared space, while consumers read data synchronously.

At the interaction scheme level, message queues recall much of tuple spaces, by representing global spaces which are fed with messages from producers. From a functional point of view, message queuing systems additionally provide transactional, timing, and ordering guarantees not necessarily considered by tuple spaces.

In the typical interaction scheme encountered in common message queuing systems, messages are concurrently pulled by consumers, in a way similar to the semantics offered by tuple spaces through the `in()` operation. These one-of-n semantics are often also referred to as *point-to-point* (PTP — see Sidebar) model. Which element is retrieved by a consumer is however not defined by the element's structure, but by the order in which the elements are stored in (i.e., received by) the queue.

Similarly to tuple spaces, producers and consumers are thus decoupled both in time and in space, but there is a lack of flow decoupling since consumers synchronously pull messages (Figure 8). Some message-queuing systems alternatively offer limited support for asynchronous message delivery. However, these asynchronous mechanisms do not scale well in the context of queuing systems because of the additional interactions needed to maintain transactional, timing, and ordering guarantees.

Summary

Traditional interaction paradigms fail to provide altogether time, space and flow decoupling required to offer a loosely form of interaction needed to build large scale distributed applications. Table 1 summarizes the decoupling properties of the considered abstractions, and Figure 9 depicts the publish/subscribe interaction paradigm.

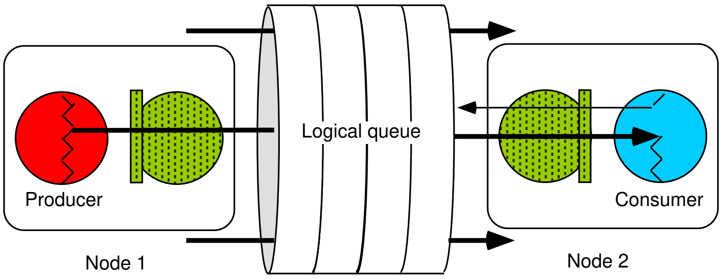


Figure 8: Message queuing — Messages are stored in a FIFO queue. Producers append messages asynchronously at the end of the queue, while consumers dequeue them synchronously at the front of the queue.

Abstraction	Space decoupling	Time decoupling	Flow decoupling
Message Passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Notifications (Observer D. Pattern)	No	No	Yes
Tuple Spaces	Yes	Yes	Producer-side
Message Queuing (Pull)	Yes	Yes	Producer-side
Publish/Subscribe	Yes	Yes	Yes

Table 1: Decoupling abilities of interaction paradigms

4 The Siblings: Publish/Subscribe Variations

Subscribers are usually interested in particular events or event patterns, and *not* in all events. The different ways of specifying the events of interest have led to several distinct event schemes. In this section we point out differences between the major schemes, namely the *topic-based*, *content-based* and *type-based* subscription schemes.

Topic-based publish/subscribe

The ancestor of all publish/subscribe schemes is based on the notion of *topics* or *subjects*, and is implemented by many industrial strength solutions. Topics represent *keywords*, to which participants can publish notifications and subscribe to. Topics are strongly similar to the notion of *groups* used for instance in the context of *group communication* [7] (e.g., for replication). This similarity is not very surprising, since some of the first systems to offer publish/subscribe interaction were based on group communication toolkits and the subscription scheme was thus inherently based on groups.

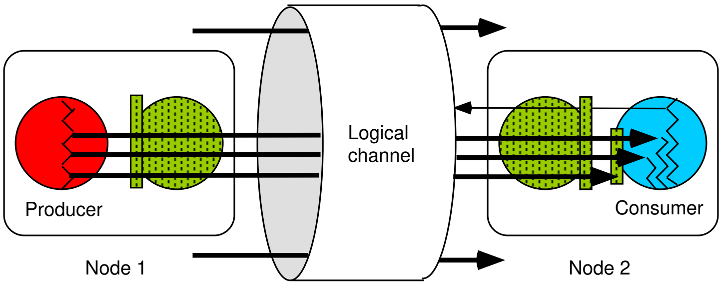


Figure 9: The publish/subscribe interaction paradigm decouples consumers and producers in terms of space, time, and flow.

Consequently, subscribing to a topic T can be viewed as becoming member of a group T, and publishing a notification for topic T translates accordingly to broadcasting that notification among the members of T. Groups and topics are similar abstractions, but correspond to two different terminologies used in different contexts: the notion of group is more often used for strong consistency between replicas of a critical component in a LAN, whereas topics are used to model large scale distributed interaction.

In practice, topic-based publish/subscribe systems introduce a programming abstraction which reflects individual topics.[5] These abstractions present interfaces similar to the event service interface we have given in Section 2, and the topic name is usually seen as an initialization argument. In other terms, every topic is viewed as an event service of its own, identified by a unique name, with an interface offering operations like `notify()` and `subscribe()`.

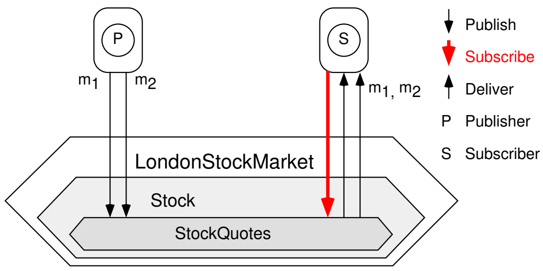
The topic abstraction is easy to understand, and enforces platform interoperability by relying only on strings as keys to divide the event space. Additions to the topic-based scheme have been proposed by various systems. The most useful improvement is the use of *hierarchies* to orchestrate topics. While group-based systems offer *flat addressing*, where groups represent disconnected event spaces, nearly all topic-based engines offer a form of *hierarchical addressing*, which permits programmers to organize topics according to containment relationships. Subscriptions can be made to any node in the hierarchy, usually implicitly involving subscriptions to all subtopics of the node. Such hierarchies are represented with a URL-like notation (introducing a hierarchy resembling much the USENET news) which is especially useful when combined with *wildcards*,[6] offering the possibility to subscribe to several topics whose names match a given set of keywords, like an entire subtree or

[5] e.g., Talarian Smart*Sockets*, Distributed Asynchronous *Collections* [3], and CORBA Event *Channels* (see Sidebar)
[6] e.g., TIBCO Rendezvous — cf. Sidebar

```
public class StockQuote implements Serializable {
  public String id;
  public String company;
  public float price;
  public int amount;
  public String traderId; }
public class StockQuoteSubscriber implements Subscriber {
  public void notify(Object o) {
     System.out.println("Got offer");
     System.out.println("Company:" + ((StockQuote)o).company);
     System.out.println("Price :" + ((StockQuote)o).price);} }

Topic quotes =
  EventService.connect("/LondonStockMarket/Stock/StockQuotes");
Subscriber sub = new StockQuoteSubscriber();
quotes.subscribe(sub);
```

(a) Sample code for topic-based subscribing



(b) Topic-based interactions

Figure 10: Topic-based publish/subscribe

a specific level in the hierarchy.

Consider the example of stock quotes disseminated to a large number of interested brokers. In a first step, we are interested in buying stocks, advertised by *stock quote* events. Such events consist of five attributes, namely a global identifier, the name of the concerned company, the price, the amount of stocks, and the identifier of the selling trader. Figure 10(a) shows how to subscribe to all stock quotes, and Figure 10(b) gives an overview of the resulting distributed interaction.

Content-based publish/subscribe

Despite improvements like hierarchical addressing facilities with wildcards, the topic-based publish/subscribe variant represents a static scheme which offers only restricted expressiveness. The *content-based* (or *property-based*) publish/subscribe variant repairs this lack by introducing a subscription scheme based on properties of the considered notifications. In other terms, events are not classified according to some pre-defined external criterion (e.g., topic name), but according to properties of the events themselves. Such properties can be internal attributes of data structures carrying notifications (e.g, Gryphon, Siena — cf. Sidebar),or meta-data associated to events (e.g, Java Messaging Service — see Sidebar).

Consumers subscribe to selective events using *name-value* pairs of properties. Such pairs are usually logically combined (and, or, etc.) to form a complex *subscription pattern*, which is translated into a *filter* applied to identify the notifications of interest for a given subscriber.[7] For subscribing, a variant of the `subscribe()` operation is provided with an additional argument representing a subscription pattern. There are several means of representing such patterns:

- **String**: The most frequently encountered scheme for expressing subscription patterns is based on strings. Filters must conform to a subscription grammar, such as SQL or the OMG's Default Filter Constraint Language. Strings are then parsed by the engine.
- **Template object**: Inspired by tuple-based matching, JavaSpaces adopts an approach based on template objects. When subscribing, a participant provides an object t, which indicates that the participant is interested in every notification which conforms to the type of t and whose attributes all match the corresponding attributes of t, except for the ones carrying a wildcard (`null`).
- **Executable code**: Subscribers provide a predicate object able to filter events at runtime. The implementation of that object is usually left to the application developer. An alternative approach based on a library of filter objects implemented using reflection is described in [2]. The subscriber describes its pattern based on *method-value* pairs, i.e., methods through which notifications are queried, and values to which invocation results are compared. The main idea behind this scheme consists in preserving encapsulation of notification objects and ensuring strong typing, while nevertheless offering good expressiveness and enforcing optimizitions, e.g., avoiding redundant queries on notification objects.

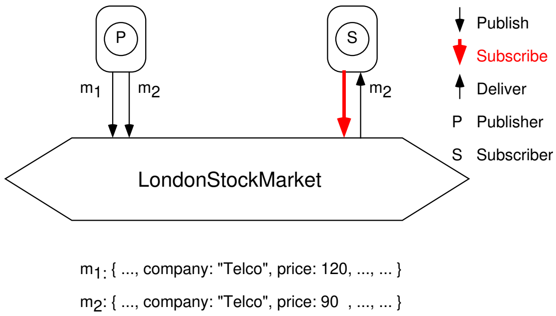
Figures 11(a) and 11(b) illustrate the use of string-based filters. The example outlines how a content-based scheme enforces a finer granularity than a static scheme based on topics. To achieve the same functionality with topics here, the subscriber would either have to filter out irrelevant events, or topics would need to be split into several subtopics — one for each company (and recursively several subtopics for different price "categories"). The first approach leads to an inefficient use of bandwidth, while the second approach results in a high number of topics and an increased risk of redundant notifications.

[7] Content-based subscription is often associated with *event correlation*, which consists in subscribing to logical combinations of elementary events. A participant which has subscribed to such a combination is only notified upon occurrence of the composite event (e.g., Cambridge event architecture CEA — see Sidebar).

```
public class StockQuote implements Serializable {
  public String id;
  public String company;
  public float price;
  public int amount;
  public String traderId; }
public class StockQuoteSubscriber implements Subscriber {
  public void notify(Object o) {
     System.out.print("Got offer for :");
     System.out.println(((StockQuote)o).price); } }

String criteria = "company == ‘Telco’ and price < 100";
Subscriber sub = new StockQuoteSubscriber();
EventService.subscribe(sub, criteria);
```

(a) Sample code for content-based subscribing



(b) Content-based interactions

Figure 11: Content-based publish/subscribe

Type-based publish/subscribe

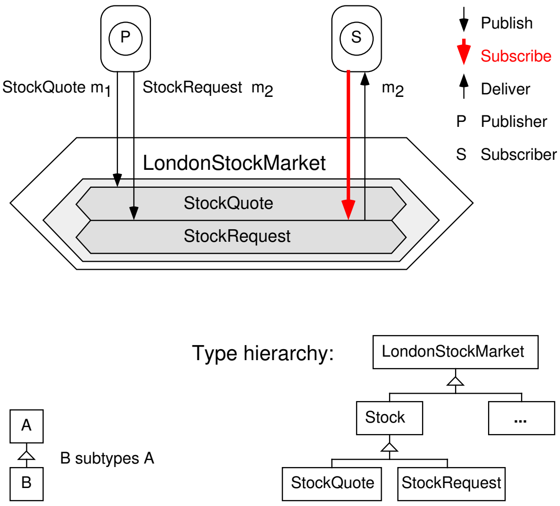
Topics usually regroup notifications that present commonalities not only in content, but also in structure. This observation has led to the idea of replacing the name-based topic classification scheme by a scheme that filters events according to their type [4]. In other terms, the notion of event *kind* is directly matched with that of event *type*. This enables a closer integration of the language and the middleware. Moreover, type safety is ensured by parameterizing the resulting abstraction interface by the type of the corresponding notifications. Genericity is thus used to enforce compile-time type checking. In contrast, the above mentioned approach chosen by JavaSpaces views the type of notifications as a dynamic property, and the resulting JavaSpace API enforces the application to perform explicit type casts. Similarly, the TAO CORBA Event Service does not view the type of a notification object as an implicit attribute.

The example in Figure 12(a) illustrates type-based subscription. The notifications for stocks can be separated into two different types, namely stock quotes (for sale), but also *stock requests*, as shown in Figure 12(b). Brokers use the latter ones to express their interest in *buying* stocks. In contrast to the stock quotes, stock requests have a range of possible prices. Subtyping can be used here to subscribe both to stock quotes and requests, without any type casts in the resulting code.

```
public class LondonStockMarket implements Serializable {
  public String id; }
public class Stock extends LondonStockMarket {
  public String company;
  public int amount;
  public String traderId; }
public class StockQuote extends Stock {
  public float price; }
public class StockRequest extends Stock {
  public float minPrice;
  public float maxPrice; }
public class StockSubscriber implements Subscriber<Stock> {
  public void notify(Stock s) {
     System.out.print("Trader " + s.traderId);
     System.out.println(" deals with " + s.company); } }

Subscriber<Stock> sub = new StockSubscriber();
EventService.subscribe<Stock>(sub);
```

(a) Sample code for type-based subscribing



(b) Type-based interactions

Figure 12: Type-based publish/subscribe

Summary

As detailed in the next section, static schemes can be implemented very efficiently, while dynamic schemes like content-based publish/subscribe offer more expressiveness but require sophisticated protocols that have higher runtime overhead. In fact, any static scheme can be expressed with a dynamic scheme like the content-based variant by considering the static key as a dynamic property of notifications. The additional overhead encourages however the use of a static scheme whenever a primary property ranges over a limited set of possible discrete values, e.g., stock quotes/requests. As outlined in [2], additional expressiveness can be achieved by enabling the application of content-based filters to a static scheme (e.g., based on topics) to express constraints on properties which are not within discrete ranges, e.g., floating point attributes like the price of a stock. In general, such a mixed scheme is very interesting, since most applications can be designed such that a coarse grain classification of notifications is sufficient in most cases, and content-based features can be used to achieve finer grain filtering when necessary.

5 The Incarnations: Implementation Issues

This section discusses some programming and algorithmic issues underlying publish/subscribe schemes, and how these issues are addressed in current systems and prototypes. Programming issues cover matters that the programmer of a given application must deal with, and algorithmic issues designate aspects like multicasting and persistence. The goal here is to discuss the main approaches with respect to the notions introduced in the previous sections, rather than giving an exhaustive enumeration of all existing concepts and respective combinations, for this has been done abundantly in [8, 9].

In this section, we focus on three major aspects of publish/subscribe middleware, namely the events, the media, and qualities of service. We discuss the different tradeoffs that result from different approaches, in terms of flexibility, reliability, scalability, and performance.

Events

Events are found in two forms: messages or invocations. In the first case, events are delivered to a subscriber through a single generic operation (e.g., `notify()`), while in the second case a subscriber receives events as invocations to its own specific operation(s).

Messages. At the lowest level, any data that goes on the network is a message. In most systems, event notifications take the form of messages, which are explicitly created by the application. Messages are generally made of a header that contains message-specific information in a generic format, and payload data that contains user-specific information. Typical header fields include message identifier, issuer, priority, or expiration time, which can be interpreted by the system or purely serve as information for the consumers. Some systems (e.g., Oracle Advanced Queuing and IBM MQSeries — see Sidebar) do not make any assumption on the type of the payload data and treat it as an opaque array of bytes. Some other systems (e.g., JMS, CORBA Notification Service) provide a set of message types (such as text messages or byte messages in the case of JMS). Finally, some systems provide self-describing messages. TIB/Rendezvous, for instance, defines a message format that does not have header information, but that allows the programmer to create his own message structure based on a set of basic types that can be structured hierarchically. The type of messages can be queried later at runtime. In most cases, messages are thus viewed as records with several fields. DACs take a different approach, in that messages can be *any* objects, provided that they are serializable (in the sense of Java). The middleware thus conforms to the application design, and not vice versa.

Invocations. At a higher level, we generally differentiate between *invocations* and *messages*. An invocation is directed to a specific object, and has well-defined semantics. The system ensures that the consumer has a matching interface for processing the invocation. The interface acts as a binding contract between the invoker and the invokee. Systems which offer invocation-style interaction along with different semantics and various addressing schemes are usually termed *messaging* systems. They offer invocations as abstraction on top of a publish/subscribe or message queuing system. While certain systems take into account return values of invocations, the typed publish/subscribe models of COM+ or the CORBA Event Service typically only consider one-way invocations. Producers invoke operations on some intermediary object (e.g., event channel) that exhibits the same interface as the actual consumers and forwards events to all registered consumers. COM+ furthermore provides a form of content-based filtering, by offering the possibility to specify values for invocation arguments in order to restrict the potential invocations.

The Media

The transmission of data between producers and consumers is the task of the middleware medium. Media can be classified according to characteristics like their architecture or the guarantees they provide for the data, such as persistence or reliability.

Architectures. The role of publish/subscribe systems is to permit the exchange of events between producers and consumers in an asynchronous manner. Asynchrony can be implemented by having producers send messages to a specific entity that stores them, and forwards them to consumers on demand. We call this approach a *centralized* architecture because of the central entity that stores and forwards messages. This centralized approach is adopted by queuing systems, like Oracle Advanced Queuing and IBM MQSeries, which are implemented through a centralized database. Applications based on such systems have strong requirements in terms of reliability, data consistency, or transactional support, but do not need a high data throughput. Examples of such applications are electronic commerce or banking applications.

Asynchrony can also be implemented by using smart communication primitives that implement store and forward mechanisms both in the producer's and consumer's processes, so that communication appears asynchronous and anonymous to the application without the need for an intermediary entity. We call this approach a *distributed* architecture because there is no central entity in the system. TIBCO Rendezvous uses a decentralized approach in which no process acts as a bottleneck or a single point of failure. Such architectures are well suited for fast and efficient delivery of transient data, which is required for applications like stock exchange or multimedia broadcasting.

Intermediate approaches implement the event notification service as a distributed network of servers such as Gryphon,Siena,or Jedi (see Sidebar).In contrast to completely decentralized systems, this approach discharges the participating processes by using dedicated servers to execute complex protocols concerned with persistence, reliability or high-availability, as well as content-based filtering and routing. There are different topologies for these servers. Jedi's *event dispatchers* are organized in a hierarchical structure, where clients can connect to any node. Subscriptions are propagated upwards the tree of servers. Such hierarchical topologies however tend to heavily load the root servers, and the failure of a server might disconnect the entire subtree. In Gryphon, a graph summarizing the common interests of subscribers is superimposed with the *message broker* graph, to avoid redundant matches. Finally, Siena implements subscription and advertisement forwarding to set the paths for notifications. *Event servers* keep track of useful information to efficiently match notifications with subscriptions afterwards. Several server topologies have been considered, each with respective advantages and shortcomings.

Dissemination. The actual transmission of data can happen in various ways. In particular, data can be sent using point-to-point communication primitives, or using hardware multicast facilities such as IP multicast. The choice of the communication mechanism depends on factors such as the target environment and the architecture of the system. Centralized approaches like certain message queuing systems are likely to use point-to-point communication primitives between producers/consumers and the centralized broker. As already mentioned, these systems focus more on strong guarantees than on high throughput and scalability.

As previously mentioned, topic-based publish/subscribe is very similar to the concept of groups. Consequently, such systems can straightforwardly benefit from the vast amount of studies on group communication [7] and the resulting protocols to disseminate notifications to subscribers. To enforce high throughput, IP multicast or a wide range of reliable multicast protocols are commonly employed.

Efficient multicast of notifications in content-based publish/subscribe systems remains an issue. Gryphon and Siena both use algorithms that deliver events to a logical network of servers in such a way that (as long as possible) a notification is propagated only to the servers that manage subscribers interested by that event. This selective notification routing inherent to content-based publish/subscribe makes the exploitation of network-level multicast primitives difficult.

Qualities of service

The guarantees provided by the medium for every message varies strongly between the different systems. Among the most common qualities of service considered in publish/subscribe, we have persistence, transactional guarantees and priorities.

Persistence. In RPC-like systems, a method invocation is by definition a transient event. The lifetime of a remote invocation is short and, if the invokee does not get a reply after a given period of time, it may re-issue the request. The situation is very different in publish/subscribe or queuing systems. Messages may be sent without generating a reply, and they may be processed hours after having been sent. The communicating parties do not control how messages are transmitted and when they are processed. Thus, the messaging system must provide guarantees not only in terms of reliability, but also in terms of durability of the information. It is not sufficient to know that a message has reached the messaging system that sits between the producers and consumers; we must get the guarantee that the message will not be lost upon failure of that messaging system.

Persistence is generally present in publish/subscribe systems that have a centralized architecture and store messages until consumers are able to process them. Queuing systems like Oracle Advanced Queuing and IBM MQSeries offer persistence using an underlying database. Distributed publish/subscribe systems generally do not offer persistence since messages are typically replicated upon emission. By sending one copy to each subscriber, the system provides some degree of fault tolerance. However, a faulty subscriber may not be able to get missed messages when recovering. TIB/Rendezvous offers a mixed approach, in which a process may listen to specific subjects, store messages on persistent storage, and re-send missed messages to recovering subscribers. The Cambridge event architectureprovides a potentially distributed event repository for event storage and efficient retrieval (with searching facilities for simple and composite events) and enables the replaying of stored sequences.

Priorities. Like persistence, message prioritization is a quality of service offered by some messaging systems. Indeed, it may be desirable to sort several messages, waiting to be processed by a consumer, in order of priority. For instance, a real-time event may require immediate reaction (e.g., failure notification) and should be processed before other messages.

Priorities affect messages that are *in transit*, i.e., not being processed. Runtime execution priorities are handled by the application scheduler and are not managed by the messaging system. In particular, this implies that two subscribers listening to the same topics may process messages in different orders because they process messages at different speeds, even though communication channels are FIFO. Priorities should be considered as a best-effort quality of service (unlike persistence).

Most publish/subscribe messaging systems (centralized or distributed) provide priorities, although the number of priorities and the way they are applied differ. Oracle Advanced Queuing, IBM MQSeries, TIB/Rendezvous and the JMS specification all integrate priorities.

Transactions. Transactions are generally used to group multiple operations in atomic blocks that are either completely executed, or not at all. In messaging systems, transactions are used to group messages into atomic units: either a complete sequence of messages is sent (received), or none of them is. For instance, a producer that publishes several semantically-related messages may not want consumers to see a partial (inconsistent) sequence of messages if it fails during emission. Similarly, a mission-critical application may want to consume one or several messages, process them, and then only commit the transaction. If the consumer fails before committing, all messages are still available for re-processing after recovery.

Due to their tight integration with databases, Oracle Advanced Queuing and IBM MQSeries provide a wide range of transactional mechanisms. JMS and TIB/Rendezvous also provide transaction support for grouping messages in the context of a single session. JavaSpaces provides lightweight transactional mechanisms to guarantee atomicity of event production and consumption. An event published in a JavaSpace in the context of a transaction is not visible outside the transaction until it is committed. Similarly, a consumed event is not removed from a JavaSpace until the enclosing transaction commits. Several events can be produced and consumed in the context of the same transaction.

6 Concluding Remarks

Publish/subscribe is more an interaction paradigm than a class of products. Publish/subscribe interaction can be build on top of other communication schemes, or by extending existing paradigms, as exemplified in Section 3 through implementations of tuple spaces or message queues. Scalability is enforced at the *abstraction level* by the publish/subscribe paradigm through the total decoupling of participants. At the *implementation level* however, scalability remains a sensitive issue. It can easily be hampered by an inappropriate architecture, in particular by building a publish/subscribe scheme on top of a system or scheme which was not designed with scalability in mind.

In short, publish/subscribe is a powerful and scalable abstraction, but an appropriate distributed infrastructure is the key to a scalable implementation of a publish/subscribe system. Unfortunately, scalability appears to be contradictory with other desirable properties. In particular, scalability seems to ask for sacrifices in terms of expressiveness (i.e., the finer subscription criteria becomes, the fewer common interests there are between subscribers, and the more complex filtering and routing becomes), but especially in terms of reliability: strong reliability guarantees involve important overheads, for instance due to logging or detecting/retransmitting missing events. Even protocols especially developed for wide-area networks still lack scalability. *Reliable Multicast Transport Protocol* (RMTP) [6] for instance is a sender-reliable (i.e., the detection of missing messages is ensured by senders) protocol based on a hierarchical distributed management of acknowledgments. The amount of traffic resulting from the acknowledgements generated by such protocols is considerable, and intrinsically limits their scalability.

Recently, probabilistic protocols have received an increasing attention since they match the decoupled and *peer-based* nature of publish/subscribe systems. Instead of providing a *complete reliability* of deterministic approaches, probabilistic multicast protocols ensure that a given notification will reach a given subscriber (or all subscribers) with a very high and quantifiable probability. The most representative protocol in this category is *pbcast* [1]. How content-based publish/subscribe systems may cope with such probabilistic protocols remains a challenging issue. Basically, appropriate trade-offs have to be defined to cope with both scalability and expressiveness in publish/subscribe systems.

While programming abstractions for publish/subscribe are plentiful, the adequate algorithms still seem to be an open issue. Much research effort will have to be invested, in particular as tribute to the unpredictability of the Internet.

References

[1] K.P. Birman, M. Hayden, O.Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[2] P.T. Eugster and R. Guerraoui. Content-Based Publish/Subscribe with Structural Reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001.

[3] P.T. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, June 2000.

[4] P.T. Eugster, R. Guerraoui, and J. Sventek. Type-Based Publish/Subscribe. Technical Report TR-DSC-2000-029, Swiss Federal Institute of Technology, Lausanne, June 2000.

[5] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, January 1985.

[6] J.C. Lin and S. Paul. A reliable multicast transport protocol. In *Proc. of IEEE INFOCOM'96*, pages 1414–1424, 1996.

[7] D. Powell. Group communication. *Communications of the ACM*, 39(4):50–97, April 1996.

[8] D. Rosenblum and A. Wolf. A design framework for internet-scale event observation and notification. In *6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering*, pages 344–360, September 1997.

[9] S. Tai and I. Rouvellou. Strategies for integrating messaging and distributed object transactions. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, pages 308–330, 2000.

Sidebar

RPC	Remote Procedure Call, see "Implementing Remote Procedure Calls", A.D. Birrel and B.J. Nelson, ACM Transactions on Computer Systems 2(1), pages 39-59, feb, 1984.
Java RMI	Java Remote Method Invocation: application of RPC to Java methods, see "Java Remote Method Invocation - Distributed Computing for Java (White Paper)", Sun Microsystems Inc., 1999, http://java.sun.com/marketing/collateral/javarmi.html.
CORBA	Common Object Request Broker Architecture: interoperable middleware platform based on remote invocations, see "The Common Object Request Broker: Architecture and Specification", OMG, October 1999, http://www.omg.org/.
CORBA Event Service	OMG's attempt to integrate asynchronous events with CORBA, see "CORBAservices: Common Object Services Specification", OMG, december 1998.
CORBA Notification Service	Follow-up of the Event Service, presents a new typed event model and QoS features, see "Notification Service Standalone Document", OMG, june 2000.
Future	Future Type Message Passing, asynchronous remote invocations. Such asynchronous invocations have appeared under different forms, like wait-by-necessity, lazy synchronization, or promises, see "A Survey of Asynchronous Remote Procedure Calls", A.L. Ananda, B.H. Tay and K.E. Koh, in ACM Operating Systems Review, 26(2), pp. 92-109, July 1992.
Observer Design Pattern	Pattern used when objects are to be notified upon changes in an object of interest (subject), see "Design Patterns, Elements of Reusable Object-Oriented Software", E. Gamma, R. Helm, R. Johnson and J. Vlissides", Addison Wesley, 1995.
MOM	Message-oriented middleware: terminology often used by message queuing systems.
PTP	Point-to-point: synomym for one-of-n used in message-queuing systems.
Oracle Advanced Queuing	Oracle's message queuing system based on the Oracle database system, see "Oracle8i Application Developer's Guide – Advanced Queuing", Oracle Corporation, 1999.
IBM MQSeries	One of the first middleware systems based on the message queuing paradigm, see "Messaging and Queuing Using the MQI", B. Blakeley, H. Harris and J.R.T. Lewis, McGraw-Hill, 1995, http://www-4.ibm.com/software/ts/mqseries/.
JMS	Java Message Service: API targeted at wrapping and unifying the diverging system, see "Java Message Service", M. Happner, R. Burridge and R. Sharma, Sun Microsystems Inc., October 1998, http://java.sun.com/products/jms/docs.html.
TIB/Rendezvous	Middleware system offering topic-based publish/subscribe, see "TIB/Rendezvous White Paper", TIBCO Inc., 1999, http://www.rv.tibco.com/.

Talarian SmartSockets	Topic-based publish/subscribe through the well-known socket abstraction, see “Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper)”, Talarian Corporation, 1999, http://www.talarian.com/.
Siena	Content-based publish/subscribe engine, see “Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service”, A. Carzaniga, D.S. Rosenblum and A.L. Wolf, in Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000), july 2000.
Gryphon	Content-based publish/subscribe based on a broker graph reflecting subscription critera, see “An efficient Multicast Protocol for Content-Based Publish-Subscribe Systems”, G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom and D.C. Sturman, in Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS’99), 1999.
CEA	Cambridge Event Architecture, system based on a *publish/register/notify* paradigm, see “Generic Support for Distributed Applications”, J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel and M. Spiteri, in IEEE Computer, march 2000.
Jedi	Java Event Dispatcher, see “Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems”, G. Cugola, E. Di Nitto and A. Fuggetta, in Proceedings of the 10th International Conference on Software Engineering (ICSE ’98), pages 261-270, april 1998.