The Many Faces of Publish/Subscribe

PATRICK TH. EUGSTER
Swiss Federal Institute of Technology in Lausanne
PASCAL A. FELBER
Institut Eurécom
RACHID GUERRAOUI
Swiss Federal Institute of Technology in Lausanne
and
ANNE-MARIE KERMARREC
Microsoft Research, Cambridge

Well-adapted to the loosely coupled nature of distributed interaction in large scale applications, the publish/subscribe communication paradigm has recently received an increasing attention. With systems based on the publish/subscribe interaction scheme, subscribers register their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events generated by publishers. Many variants of the paradigm have recently been proposed, 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 entities in time, space and synchronization. We use these three decoupling dimensions to better identify commonalities and divergences with traditional interaction paradigms. The many 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.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

General Terms: Design

Additional Key Words and Phrases: distribution, interaction, publish/subscribe

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 greatly vary throughout the lifetime of the system. These constraints visualize the demand for more flexible communi-

Authors' addresses: P.Th. Eugster, Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland. P.A. Felber, Institut Eurécom, 2229 route des Crêtes, 06904 Sophia Antipolis, France. R. Guerraoui, Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland. A.-M. Kermarrec, Microsoft Research Ltd., 7 J J Thomson Ave, Cambridge CB3 0FB, UK.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2003 ACM 0000-0000/2003/0000-0001 $5.00

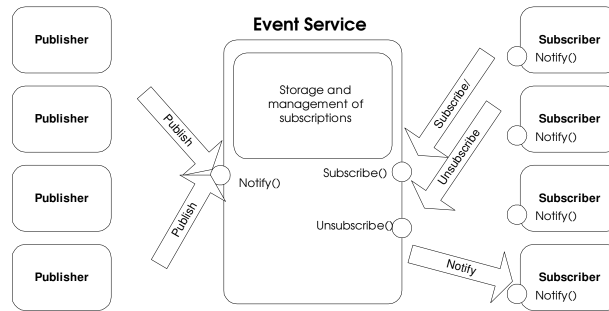

Fig. 1. A simple object-based publish/subscribe system

cation 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 middleware infrastructure, based on an adequate communication scheme.

The *publish/subscribe* interaction scheme is 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 are subsequently notified of any event, generated by a publisher, which matches their registered interest. An event is asynchronously propagated to all subscribers that registered interest in that given event. The strength of this event-based interaction style lies in the full decoupling in *time*, *space* and *synchronization* between publishers and subscribers. Many industrial systems and research prototypes support this style of interaction, and there are several prominent research efforts on novel forms of publish/subscribe interaction schemes. However, because of the multiplicity of these systems and prototypes, it is difficult to capture their commonalities and draw sharp lines 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 *synchronization* 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.

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 subsequently of any event, generated by a publisher, that matches 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

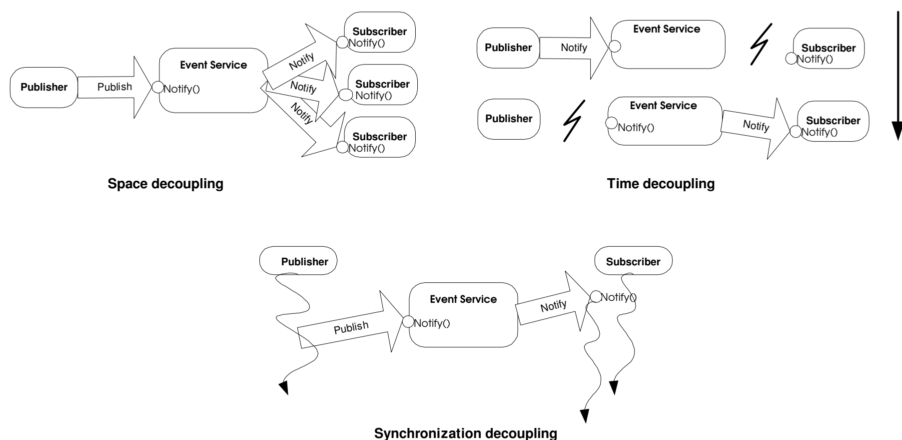

Fig. 2. Space, time and synchronization decoupling with the publish/subscribe paradigm

that bus. This information is typically denote by the term *event* and the act of delivering it by the term *notification*.

The basic system model for publish/subscribe interaction (Figure 1) relies on an event notification service providing storage and management for subscriptions and efficient delivery of events. Such an event service represents a neutral mediator between publishers, acting as producers of events, and subscribers, acting as consumers of events. 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 generate an event, a publisher typically calls a `notify()` (or `publish()`) operation. The event service propagates the event to all relevant subscribers; it can thus be viewed as a proxy for the subscribers. Note that *every* subscriber will receive an event for *every* event conforming to its interest (obviously, failures might prevent subscribers from receiving some events). Publishers also often have the ability to advertise the nature of their future events through an `advertise()` operation. 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 when a new type of information becomes available.

The decoupling that the event service provides between publishers and subscribers can be decomposed along the 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 through the event service. The publishers do not usually hold references to the subscribers, neither do they know how many of these subscribers are participating in the interaction. Similarly, subscribers do not usually hold references to the publishers, neither do they know how many of these publishers are participating in the interaction.

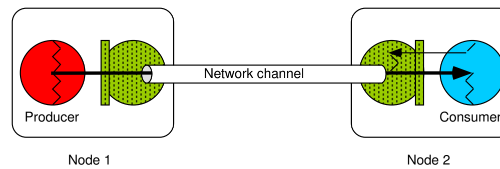

Fig. 3. Message passing interaction—The producer sends messages asynchronously through a communication channel (previously set up for that purpose). The consumer receives messages by listening synchronously on that channel.

— **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.

— **Synchronization decoupling**: publishers are not blocked while producing events, and subscribers can get asynchronously notified (through a callback) of the occurrence of an event while performing some concurrent activity. The production and consumption of messages do not happen in the main flow of control of the publishers and subscribers, and do not therefore happen in a synchronous manner.

Decoupling 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, and makes the resulting communication infrastructure well adapted to distributed environments that are asynchronous by nature, such as mobile environment [Huang and Garcia-Molina 2001].

Complementary classifications of the interaction models of distributed information systems have been proposed in the literature. Franklin and Zdonik Franklin and Zdonik [1997] classify dissemination-based systems according to their data delivery mechanisms: push vs. pull, aperiodic vs. periodic, and unicast vs. 1-to-N. Push-based information systems have been studied extensively [Hauswirth and Jazayeri 1999; Hauswirth 1999]. Similar characterizations are used in software engineering [Sullivan and Notkin 1990; Garlan and Notkin 1991] and coordination models [Papadopoulos and Arbab 1998].

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. They stand at different abstraction levels and are not easy to compare. Nevertheless, we overview below their commonalities with publish/subscribe systems and emphasize their inability to fully decouple communication between participants.

3.1 Message passing

Message passing can be viewed as the ancestor of distributed interactions. Message passing represents a low-level form of distributed communication, in which participants communicate by simply sending and receiving messages. Although complex interaction schemes are still built on top of such primitives, message passing is nowadays rarely used directly for developing distributed applications, since

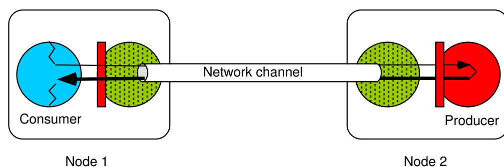

Fig. 4. RPC and derivatives—The producer performs a synchronous call, which is processed asynchronously by the consumer.

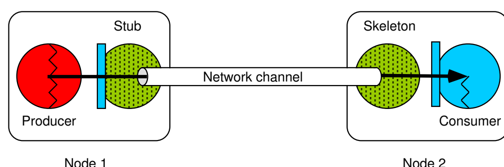

Fig. 5. Decoupling synchronization with asynchronous remote invocation—The producer does not expect a reply.

physical addressing and data marshaling, and sometimes even flow control (e.g., retransmission), become visible to the 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 recipient of a message is known to the sender.

3.2 RPC

One of the most widely used forms of distributed interaction is the remote invocation, an extension of the notion of "operation invocation" to a distributed context. This type of interaction has first been proposed in the form of *Remote Procedure Call* (RPC) [Birrell and Nelson 1983; Tay and Ananda 1990] for procedural languages, and has been straightforwardly applied to object-oriented contexts in the form of remote method invocations, e.g., in Java RMI [Sun 2000], CORBA [OMG 2002a], Microsoft DCOM [Horstmann and Kirtland 1997; Chung et al. 1998].

By making remote interactions appear the same way as local interactions, the RPC model and its derivatives make distributed programming very easy. This explains their tremendous popularity in distributed computing. Distribution cannot, however, be made completely transparent to the application, because it gives rise to further types of potential failures (e.g., communication failures) that have to be dealt with explicitly. As shown in Figure 4, RPC differs from publish/subscribe in terms of coupling: the synchronous nature of RPC introduces a strong time, synchronization (on the consumer side[1]), and also space coupling (since an invoking object holds a remote reference to each of its invokees).

Several attempts have been made to remove synchronization coupling in remote and avoid blocking the caller thread while waiting for the reply of a remote invocation. A first variant consists in providing a special flavor of asynchronous invocation for remote methods that have no return values, as shown in Figure 5. For instance, CORBA provides a special *oneway* modifier that can be used to specify

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

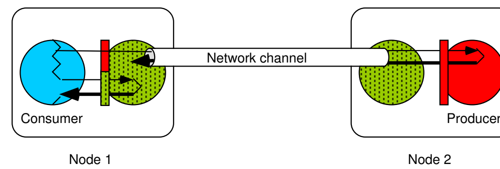

Fig. 6. Decoupling synchronization with future remote invocation—The producer is not blocked and can access the reply later when it becomes available.

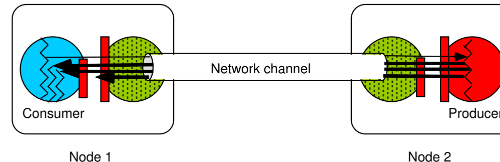

Fig. 7. Notifications—Producers and consumers communicate using asynchronous invocations flowing in both directions.

such methods [OMG 2002a]. This approach leads to invocations with weak reliability guarantees because the sender does not receive success or failure notifications (this type of interaction is often called *fire-and-forget*). The second, less restrictive variant supports return values, but does not make them directly available to the calling thread. Instead, the result of a remote invocation is a handle through which the actual return values will be accessed when needed. With this approach, known as *future* or *future type message passing* [Yonezawa et al. 1987; Ananda et al. 1992] or *wait-by-necessity* [Caromel 1993], the invoking thread can continue processing and request the return value later, thanks to the handle (Figure 6).

3.3 Notifications

In order to achieve synchronization 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 extended to return several replies by having the server make several callbacks to the client. Such notification-based interaction is widely used to ensure consistency of Web caches [Wessels 1995]: upon download of Web contents, Web proxies receive a *promise* to be notified if any change occurs at the Web server. This implements a limited form of publish/subscribe interaction in which Web proxies act as subscribers and the Web server as the publisher.

This type of interaction—where subscribers register their interest directly with publishers, which manage subscriptions and send events—corresponds to the so-called *observer design pattern* [Gamma et al. 1995] (Figure 7). It is generally implemented using asynchronous invocations in order to enforce synchronization decoupling. Although publishers notify subscribers asynchronously, they both remain coupled in time and in space. Furthermore the communication management is left to the publisher and can become burdensome as the system grows in size.

3.4 Shared spaces

The *distributed shared memory* (DSM) paradigm [Li and Hudak 1989; Tam et al. 1990] provides hosts in a distributed system with the view of a common shared space across disjoint address spaces, in which synchronization and communication

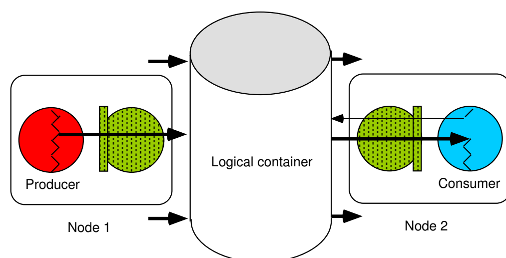

Fig. 8. Shared space—Producers insert data asynchronously into the shared space, while consumers read data synchronously.

between participants take place through operations on shared data. The notion of *tuple space* has been originally integrated at the language level in Linda [Gelernter 1985], and provides a simple and powerful abstraction for accessing shared memory. 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/removal of tuples into/from the tuple space. Three main operations can be performed: `out()` to export a tuple into a tuple space, `in()` to import (and remove) a tuple from the tuple space, and `read()` to read (without removing) 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 can be used to implement *one-to-n* message delivery (a given tuple can be read by all consumers). Unlike the publish/subscribe paradigm, the DSM model does not provide synchronization decoupling because consumers pull new tuples from the space in a synchronous style (Figure 8). This limits the scalability of the model due to the required synchronization between the participants. To compensate the lack of synchronization decoupling, some modern tuple space systems like JavaSpaces [Sun 2002], TSpaces [Lehman et al. 1999], and WCL [Rowstron 1998] extend the Linda tuple space model with asynchronous notifications.

A similar communication abstraction, called *rendezvous*, has been introduced in the Internet Indirection Infrastructure (I3) [Stoica et al. 2002]. Instead of explicitly sending a packet to a destination, each packet is associated with an identifier; this identifier is then used by the receiver to obtain delivery of the packet. This level of indirection decouples the act of sending from the act of receiving.

3.5 Message queuing

Message queuing [Blakeley et al. 1995] is a more recent alternative for distributed interaction. In fact, the term message queuing is often used to refer to a family of products (e.g., [Corporation 1995; Houston 1998; DEC 1994; Oracle 2002]) 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) [Banavar et al. 1999].

At the interaction level, message queues recall much of tuple spaces: queues can

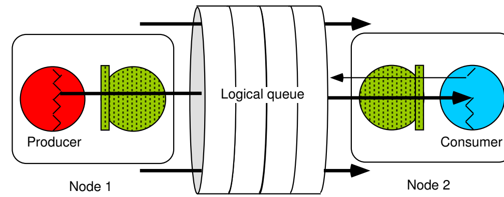

Fig. 9. Message queuing—Messages are stored in a FIFO queue. Producers append messages asynchronously at the end of the queue, while consumers dequeue them synchronously at the front of the queue.

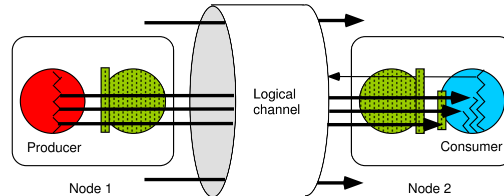

Fig. 10. The publish/subscribe interaction paradigm decouples consumers and producers in terms of space, time, and synchronization.

Table I. Decoupling abilities of interaction paradigms

Abstraction	Space decoupling	Time decoupling	Synchronization 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 Pattern)	No	No	Yes
Tuple Spaces	Yes	Yes	Producer-side
Message Queuing (Pull)	Yes	Yes	Producer-side
Publish/Subscribe	Yes	Yes	Yes

be seen as 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 message queuing systems, messages are concurrently pulled by consumers with *one-of-n* semantics similar to those offered by tuple spaces through the `in()` operation (Figure 9). These interaction model is often also referred to as *Point-To-Point* (PTP) queuing. Which element is retrieved by a consumer is not defined by the element's structure, but by the order in which the elements are stored in the queue (generally FIFO or priority-based order).

Similarly to tuple spaces, producers and consumers are decoupled in both time and space. As consumers synchronously pull messages, message queues do not provide synchronization decoupling. Some message-queuing systems offer limited support for asynchronous message delivery, but these asynchronous mechanisms do not scale well to large populations of consumers because of the additional interactions needed to maintain transactional, timing, and ordering guarantees.

3.6 Summary

Traditional interaction paradigms essentially differ from publish/subscribe communication (Figure 10) by their limited support for time, space and synchronization decoupling. Table I summarizes the decoupling properties of the aforementioned

communication models.

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 subscription schemes. In this section we compare the two most widely used schemes, namely *topic-based* and *content-based* publish/subscribe, as well the recently proposed *type-based* subscription scheme.

4.1 Topic-based publish/subscribe

The earliest publish/subscribe scheme is based on the notion of *topics* or *subjects*, and is implemented by many industrial strength solutions (e.g., [Altherr et al. 1999; Corporation 1999; Skeen 1998; TIBCO 1999]). It extends the notion of channels, used to bundle communicating peers, with methods to characterize and classify event content. Participants can publish events and subscribe to individual topics, which are identified by *keywords*. Topics are strongly similar to the notion of *groups*, as defined in the context of *group communication* [Powell 1996] and often used for replication [Birman 1993]. This similarity is not surprising, since some of the first systems to offer publish/subscribe interaction were based on the Isis [Birman et al. 1990] group communication toolkit and the subscription scheme was thus inherently based on groups. Consequently, subscribing to a topic T can be viewed as becoming member of a group T, and publishing an event on topic T translates accordingly into broadcasting that event among the members of T. Although groups and topics are similar abstractions, they are generally associated to different application domains: groups are used for maintaining strong consistency between the replicas of a critical component in a LAN, whereas topics are used to model large scale distributed interactions.

In practice, topic-based publish/subscribe systems introduce a programming abstraction which maps individual topics to distinct communication channels. They present interfaces similar to those of the event service of Section 2, and the topic name is usually specified as an initialization argument. Every topic is viewed as an event service of its own, identified by a unique name, with an interface offering `notify()` and `subscribe()` operations.

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 modern topic-based engines offer a form of *hierarchical addressing*, which permits programmers to organize topics according to containment relationships. A subscription made to some node in the hierarchy implicitly involves subscriptions to all the subtopics of that node. Topic names are generally represented with a URL-like notation and introduce a hierarchy very similar to the USENET news. Most systems allow topic names to contain *wildcards*, first introduced in TIBCO Rendezvous [TIBCO 1999], which offer the possibility to subscribe and publish to several topics whose names match a given set of keywords, like an entire subtree or a specific level in the hierarchy.

```
public class StockQuote implements Serializable {
  public String id, company, trader;
  public float price;
  public int amount;
}
public class StockQuoteSubscriber implements Subscriber {
  public void notify(Object o) {
    if (((StockQuote)o).company == 'TELCO' && ((StockQuote)o).price < 100)
      buy();
  }
}
// ...
Topic quotes = EventService.connect("/LondonStockMarket/Stock/StockQuotes");
Subscriber sub = new StockQuoteSubscriber();
quotes.subscribe(sub);
```

Fig. 11. Sample code for topic-based publish/subscribe.

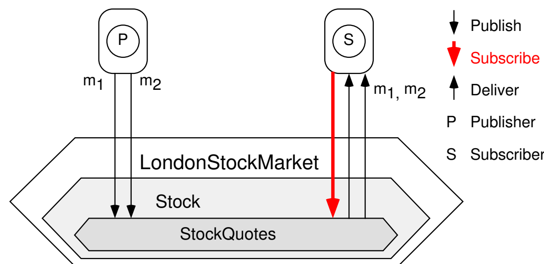

Fig. 12. Topic-based publish/subscribe interactions.

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: a global identifier, the name of the company, the price, the amount of stocks, and the identifier of the selling trader. Figure 11 shows how to subscribe to all stock quotes, and Figure 12 gives an overview of the resulting distributed interaction.

4.2 Content-based publish/subscribe

Despite improvements like hierarchical addressing facilities and wildcards, the topic-based publish/subscribe variant represents a static scheme which offers only limited expressiveness. The *content-based* (or *property-based* [Rosenblum and Wolf 1997]) publish/subscribe variant improves on topics by introducing a subscription scheme based on the actual content of the considered events. In other terms, events are not classified according to some pre-defined external criterion (e.g., topic name), but according to the properties of the events themselves. Such properties can be internal attributes of data structures carrying events, as in Gryphon [Banavar et al. 1999], Siena [Carzaniga et al. 2000], Elvin [Segall et al. 2000], and Jedi [Cugola et al. 2001], or meta-data associated to events, as in the Java Messaging Service [Hapner et al. 2002].

Consumers subscribe to selective events by specifying filters using a subscription language. The filters define constraints, usually in the form of *name-value*

```
public class StockQuote implements Serializable {
  public String id, company, trader;
  public float price;
  public int amount;
}
public class StockQuoteSubscriber implements Subscriber {
  public void notify(Object o) {
    buy();      // company == 'TELCO' and price < 100
  }
}
// ...
String criteria = ("company == 'TELCO' and price < 100");
Subscriber sub = new StockQuoteSubscriber();
EventService.subscribe(sub, criteria);
```

Fig. 13. Sample code for content-based publish/subscribe.

pairs of properties and basic comparison operators ($=$, $<$, $\leq$, $>$, $\geq$), which identify valid events. Constraints can be logically combined (and, or, etc.) to form complex *subscription patterns*. Some systems, like the Cambridge Event Architecture (CEA) [Bacon et al. 2000], also provide for *event correlation*: participants can subscribe to logical combinations of elementary events and are only notified upon occurrence of the composite events. Subscription patterns are used to identify the events of interest for a given subscriber and propagate events accordingly. For subscribing, a variant of the `subscribe()` operation is provided by the event service, with an additional argument representing a subscription pattern. There are several means of representing such patterns:

— **String**: Subscription patterns are most frequently expressed using strings. Filters must conform to a subscription grammar, such as SQL [Hapner et al. 2002; Oracle 2002; Lewis 1999], OMG's Default Filter Constraint Language [OMG 2002b], XPath [Altinel and Franklin 2000; Chan et al. 2002; Diao et al. 2002], or some proprietary language [Banavar et al. 1999; Carzaniga et al. 2001; Segall and Arnold 1997]. Strings are then parsed by the engine.

— **Template object**: Inspired by tuple-based matching, JavaSpaces [Freeman et al. 1999] adopts an approach based on template objects. When subscribing, a participant provides an object t, which indicates that the participant is interested in every event that 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 [Eugster and Guerraoui 2001]. Executable code is not widely used in practice because the resulting filters are extremely hard to optimize, and they must generally be applied to each event sequentially, leading to poor scalability.

Figures 13 and 14 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, the subscriber would either

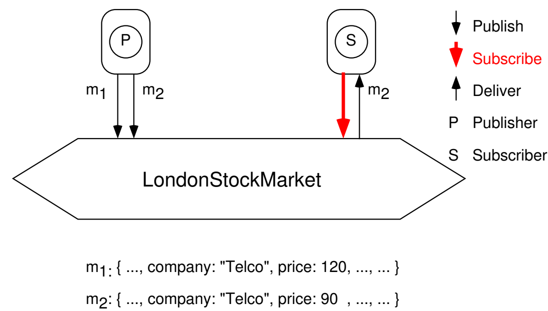

Fig. 14. Content-based publish/subscribe interactions.

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 events.

4.3 Type-based publish/subscribe

Topics usually regroup events 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 model by a scheme that filters events according to their type [Eugster et al. 2001]. 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 can be ensured at compile-time by parameterizing the resulting abstraction interface by the type of the corresponding events (without any type cast in the resulting code). In contrast, the aforementioned template-based approach of JavaSpaces [Freeman et al. 1999] considers the type of events as a dynamic property, and the resulting JavaSpace API enforces the application to perform explicit type casts. Similarly, the TAO CORBA Event Service [Harrison et al. 1997] does not view the type of an event object as an implicit attribute.

The example in Figure 15 illustrates type-based subscription. Stock events can be split into two distinct types: *stock quotes* (for sale) and *stock requests*, as shown in Figure 16. Brokers use stock requests to express their interest in *buying* stock. In contrast to quotes, requests have a range of possible prices. Subtyping can be used to subscribe to both stock quotes and requests.

It is important to notice that type-based publish subscribe can lead to a natural description of content-based filtering through public members of the considered event type, while ensuring the encapsulation of these events. This can is achieved in our example of Figure 15 by declaring only private data members and enforcing their access through public methods.

4.4 Summary

There exists several variants for designing publish/subscribe systems, which offer different degrees of expressiveness and, as we shall see in the next section, different performance overhead. Topic-based publish/subscribe is rather static and primitive,

```
public class LondonStockMarket implements Serializable {
  public String getId() {...}
}
public class Stock extends LondonStockMarket {
  public String getCompany() {...}
  public String getTrader() {...}
  public int getAmount() {...}
}
public class StockQuote extends Stock {
  public float getPrice() {...}
}
public class StockRequest extends Stock {
  public float getMinPrice() {...}
  public float getMaxPrice() {...}
}
public class StockSubscriber implements Subscriber<StockQuote> {
  public void notify(StockQuote s) {
    if (s.getCompany() == 'TELCO' && s.getPrice() < 100)
      buy();
  }
}
// ...
Subscriber<StockQuote> sub = new StockSubscriber();
EventService.subscribe<StockQuote>(sub);
```

Fig. 15. Sample code for type-based publish/subscribe.

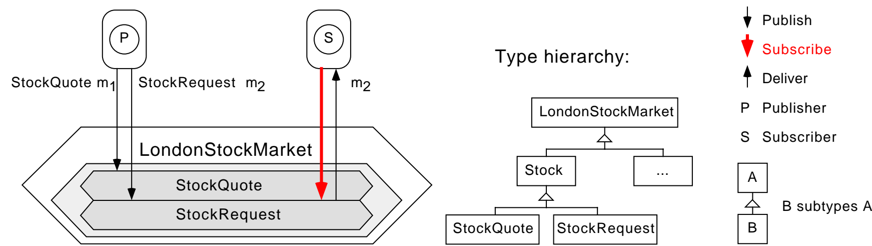

Fig. 16. Type-based publish/subscribe interactions.

but can be implemented very efficiently. On the other hand, content-based publish/subscribe is highly expressive, but requires sophisticated protocols that have higher runtime overhead. Because of this additional overhead, one should generally prefer a static scheme whenever a primary property ranges over a limited set of possible discrete values, e.g., stock quotes/requests. As outlined in [Eugster and Guerraoui 2001], additional expressiveness can be achieved by applying content-based filters in the context of statically-configured topics, in particular types, to express constraints on properties that are not within discrete ranges (e.g., stock prices).

5. THE INCARNATIONS: IMPLEMENTATION ISSUES

This section discusses some implementation issues underlying publish/subscribe schemes, and how these issues are addressed in current systems and prototypes. We focus on three major aspects of publish/subscribe middleware: the events, the media, and qualities of service, in the context of the classification introduced in the previous sections. Furthermore, we discuss the different tradeoffs that result from different approaches, in terms of flexibility, reliability, scalability, and performance. Additional details on specific implementation issues of publish/subscribe systems can be found in [Rosenblum and Wolf 1997; Banavar et al. 1999; Tai and Rouvellou 2000].

5.1 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 events trigger the execution of specific operations of the subscriber.

5.1.1 *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., IBM MQSeries [Lewis 1999] and Oracle Advanced Queuing [Oracle 2002]) 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 [Hapner et al. 2002], CORBA Notification Service [OMG 2002b]) provide a set of message types, such as text or XML messages. Finally, some systems provide self-describing messages. TIBCO Rendezvous [TIBCO 1999], for instance, defines a message format that does not have header information, but 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. DAC [Eugster et al. 2000] and JMS [Hapner et al. 2002] even support object messages, where the event can be any serializable Java object. In most cases, messages are viewed as records with several fields.

5.1.2 *Invocations.* At a higher level, we generally differentiate between *invocations* and *messages.* An invocation is directed to a specific type of object, and has well-defined semantics. The system ensures that all consumers have a matching interface for processing the invocation. The interface acts as a binding contract between the invoker and the invokees. Systems which offer invocation-style interaction along with different semantics and various addressing schemes are usually termed *messaging* systems. They incorporate additional logic on top of a publish/subscribe or message queuing system to transform low-level messages into invocations to methods of the subscribers, which must all be of the same type. While certain systems take into account return values of invocations, the typed publish/subscribe models of COM+ [Sessions 1997] or the CORBA Event Service [OMG 2001] typ-

ically 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.

5.2 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.

5.2.1 *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 approach is adopted by queuing systems like IBM MQSeries [Lewis 1999] and Oracle Advanced Queuing [Oracle 2002], which are built on top of 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 [TIBCO 1999] 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.

An intermediate approach, adopted for instance by Gryphon [Banavar et al. 1999], Siena [Carzaniga et al. 2000], and Jedi [Cugola et al. 2001], consists in implementing the event notification service as a distributed network of servers. In contrast to completely decentralized systems, this approach discharges the participating processes by using dedicated servers to execute the complex protocols required for 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 tend, however, 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. Siena uses subscription and advertisement forwarding to set the paths for notifications. *Event servers* keep track of useful information to efficiently match events with subscriptions. Several server topologies have been

considered, each with respective advantages and shortcomings.

5.2.2 *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 like IP multicast [Deering]. 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. Topic-based publish/subscribe systems can straightforwardly benefit from the vast amount of studies on group communication [Powell 1996] and the resulting protocols to disseminate events to subscribers. To ensure high throughput, IP multicast or a wide range of reliable multicast protocols [Floyd et al. 1997; Holbrook et al. 1995; Lin and Paul 1996; Castro et al. 2002; Banerjee et al. 2002; Ratnasamy et al. 2001; Zhuang et al. 2001] are commonly employed.

Efficient multicast of events in content-based publish/subscribe systems remains an issue. Gryphon and Siena both use algorithms [Aguilera et al. 1999; Carzaniga et al. 2001] that deliver events to a logical network of servers in such a way that an event is propagated only to the servers that manage subscribers interested by that event. The performance of such dissemination-based systems is strongly affected by the cost of event filtering on each of the server, which directly depends on the number of subscription in the system. Highly-efficient and scalable algorithms have been recently proposed for filtering data in publish/subscribe systems [Altinel and Franklin 2000; Pereira et al. 2000; Fabret et al. 2001; Campailla et al. 2001; Chan et al. 2002; Diao et al. 2002]. The problem of aggregating subscriptions to increase the filtering speed at each server, at the price of a small loss in precision, has been studied in [Chan et al. 2002]. Irrespective of the filtering techniques, the selective event routing inherent to content-based publish/subscribe makes the exploitation of network-level multicast primitives difficult.

5.3 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.

5.3.1 *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 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 IBM MQSeries [Lewis 1999] and Oracle Advanced Queuing [Oracle 2002] offer persistence using an underlying database. Distributed publish/subscribe systems do not generally offer persistence since messages are directly sent by the producer to all subscribers. Unless the producer keeps a copy of each message, a faulty subscriber may not be able to get missed messages when recovering. TIBCO Rendezvous [TIBCO 1999] 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 Architecture [Bacon et al. 2000] provides a potentially distributed event repository for event storage and efficient retrieval (with searching facilities for simple and composite events) that enables the replaying of stored sequences of events.

5.3.2 *Priorities.* Like persistence, message prioritization is a quality of service offered by some messaging systems. Indeed, it may be desirable to sort the 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. IBM MQSeries [Lewis 1999], Oracle Advanced Queuing [Oracle 2002], TIBCO Rendezvous [TIBCO 1999] and the JMS specification [Hapner et al. 2002] all support priorities.

5.3.3 *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, IBM MQSeries [Lewis 1999] and Oracle Advanced Queuing [Oracle 2002] provide a wide range of transactional mechanisms. JMS [Hapner et al. 2002] and TIBCO Rendezvous [TIBCO 1999] also provide transaction support for grouping messages in the context of a single session. JavaSpaces [Freeman et al. 1999] 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.

5.3.4 *Reliability.* Reliability is an important feature of distributed information systems. It is often necessary to have strong guarantees about the reliable delivery of information to one or several distributed entities. Because of the loose synchronization between producers and consumers of information, implementing reliable event propagation ("guaranteed delivery") is challenging.

Centralized publish/subscribe systems generally use reliable point-to-point channels to communicate with publishers and subscribers, and keep copies of events on stable storage. Events are therefore reliably delivered to all subscribers, although a failure of the centralized event broker may delay delivery.

Systems based on an overlay network of distributed event brokers often use reliable protocols to propagate events to all or a subset of the brokers. Protocols based on group communication [Powell 1996] and reliable application-layer multicast [Floyd et al. 1997; Holbrook et al. 1995; Lin and Paul 1996; Castro et al. 2002; Banerjee et al. 2002; Ratnasamy et al. 2001; Zhuang et al. 2001] are good candidates as they are resilient to the failure of some of the brokers. Individual publishers and subscribers generally communicate with the nearer broker using point-to-point communication channels.

Finally, systems that let publishers and subscriber communicate directly with each other, such as TIBCO Rendezvous [TIBCO 1999], also use lightweight reliable multicast protocols. As events are generally not kept in the system for failed or disconnected (time-decoupled) subscribers, guaranteed delivery must be implemented by deploying dedicated processes that store events and replay them to requesting subscribers.

6. CONCLUDING REMARKS

Publish/subscribe is a distributed interaction paradigm well adapted to the deployment of scalable and loosely-coupled systems. To survey and compare distributed event-based abstractions, we have introduced a classification based on three dimensions: the decoupling in *time*, *space* and *synchronization* between producers and consumers of information. Decoupling is a desirable property because it enforces scalability at the *abstraction level*, by allowing participants to operate independently of one another. At the *implementation level* however, scalability remains a sensitive issue, because publish/subscribe interaction can be build on top of various communication substrates and can easily be hampered by an inappropriate architecture, in particular when publish/subscribe systems are built on top of infrastructures that were not designed with scalability in mind.

Scalability also often conflicts with other desirable properties. For instance, highly expressive and selective subscriptions require complex and expensive filtering and routing algorithms, and thus limit scalability. Similarly, strong reliability guarantees involve important overheads, because events must be logged, and missed events must be detected and retransmitted. Even protocols developed especially for wide-area networks, such as the sender-reliable *Reliable Multicast Transport Proto-*

col (RMTP) [Lin and Paul 1996], do not scale well to large numbers of subscribers because of the considerable amount of traffic resulting from message acknowledgments.

Recently, probabilistic protocols have received increasing attention since they match the decoupled and *peer-based* nature of publish/subscribe systems. Instead of providing *deterministic* (guaranteed) reliability, probabilistic multicast protocols ensure that a given event will reach all subscribers with a very high and quantifiable probability [Birman et al. 1999]. Integration of such probabilistic protocols in content-based publish/subscribe systems remains a challenging issue.

While programming abstractions for publish/subscribe are plentiful, designing appropriate algorithms for deploying such systems in a large scale is still an open issue, and trade-offs must be dealt with to cope with scalability, expressiveness and quality of service. Significant research efforts remain to be invested, in particular as tribute to the unpredictability of the Internet.

REFERENCES

Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M., and Chandra, T. D. 1999. Matching Events in a Content-based Subscription System. In *Proc. of ACM PODC*. Atlanta, GA, 53–61.

Altherr, M., Erzberger, M., and Maffeis, S. 1999. iBus – a software bus middleware for the Java platform. In *International Workshop on Reliable Middleware Systems*. 43–53.

Altinel, M. and Franklin, M. 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB 2000)*. 53–64.

Ananda, A., Tay, B., and Koch, K. 1992. A Survey of Asynchronous Remote Procedure Calls. *ACM Operating Systems Review 26,* 2 (July), 92–109.

Bacon, J., Moody, K., Bates, J., R.Hayton, Ma, C., McNeil, A., Seidel, O., and Spiteri, M. 2000. Generic support for distributed applications. *IEEE Computer*.

Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., and Sturman, D. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*.

Banavar, G., Chandra, T., Strom, R., and Sturman, D. 1999. A case for message oriented middleware. In *13th International Symposium on Distributed Computing (DISC 99)*. 1–18.

Banerjee, S., Bhattacharjee, B., and Kommareddy, C. 2002. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM*.

Birman, K. 1993. The process group approach to reliable distributed computing. *Communications of the ACM 36,* 12 (Dec.), 36–53.

Birman, K., Cooper, R., Joseph, T., Marzullo, K., Makpangou, M., Kane, K., Schmuck, F., and Wood, M. 1990. *The Isis System Manual*. Dept. of Computer Science, Cornell University.

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

Birrell, A. D. and Nelson, B. J. 1983. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*. Bretton Woods, NH, 3.

Blakeley, B., Harris, H., and Lewis, J. 1995. *Messaging and Queuing Using the MQI*. McGraw-Hill.

Campailla, A., Chaki, S., Clarke, E., Jha, S., and Veith, H. 2001. Efficient filtering in publish-subscribe systems using binary decision. In *International Conference on Software Engineering*. 443–452.

Caromel, D. 1993. Towards a method of object-oriented concurrent programming. *Communications of the ACM 36*, 90–102.

Carzaniga, A., Rosenblum, D., and Wolf, A. 2000. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*.

Carzaniga, A., Rosenblum, D., and Wolf, A. 2001. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Trans. on Computer Systems 19,* 3 (August), 332–383.

Castro, M., Druschel, P., Kermarrec, A.-M., and Rowstron, A. 2002. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC) 20,* 8 (oct).

Chan, C.-Y., Fan, W., Felber, P., Garofalakis, M., and Rastogi, R. 2002. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*. Hong Kong, China.

Chan, C.-Y., Felber, P., Garofalakis, M., and Rastogi, R. 2002. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*. San Jose, CA.

Chung, P., Huang, Y., Yajnik, S., Liang, D., Shih, J., Wang, C.-Y., and Wang, Y. 1998. DCOM and CORBA side by side, step by step, and layer by layer. *C++ Report 10,* 1 (Jan).

Corporation, I. 1995. MQSeries: An introduction to messaging and queuing. Tech. Rep. GC33-0805-01, IBM Corporation. Jul.

Corporation, T. 1999. *Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper)*. http://www.talarian.com/.

Cugola, G., Nitto, E. D., and Fugetta, A. 2001. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering 27,* 9 (Sept.), 827–850.

DEC. 1994. *DECMessageQ: Introduction to Message Queuing*.

Deering, S. Host extension for ip multicast. IETF RFC 1112.

Diao, Y., Fischer, P., Franklin, M., and To, R. 2002. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*. San Jose, CA.

Eugster, P. and Guerraoui, R. 2001. Content-Based Publish/Subscribe with Structural Reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*.

Eugster, P., Guerraoui, R., and Damm, C. 2001. On Objects and Events. In *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming Systems Languages and Applications*. ACM Press, 254–269.

Eugster, P., Guerraoui, R., and Sventek, J. 2000. Distributed Asynchronous Collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*.

Fabret, F., Jacobsen, H., Llirbat, F., Pereira, J., Ross, K., and Shasha, D. 2001. Filtering Algorithms and Implementations for Very Fast Publish/Subscribe Systems. In *Proc. of ACM SIGMOD*. Santa Barbara, California, 115–126.

Floyd, S., Jacobson, V., liu, C., McCanne, S., and Zhang, L. 1997. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transaction on networking*, 784–803.

Franklin, M. and Zdonik, S. 1997. A framework for scalable dissemination-based systems. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. 94–105.

Freeman, E., Hupfer, S., and Arnold, K. 1999. *JavaSpaces Principles, Patterns, and Practice*. Addison Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley.

Garlan, D. and Notkin, D. 1991. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Formal Software Development Methods*. Lecture Notes in Computer Science, vol. 551. Springer-Verlag, 31–44.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 80–112.

HAPNER, M., BURRIDGE, R., SHARMA, R., FIALLI, J., AND STOUT, K. 2002. *Java Message Service*. Sun Microsystems Inc.

HARRISON, T., LEVINE, D., AND SCHMIDT, D. 1997. The design and performance of a real-time CORBA event service. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. 184–200.

HAUSWIRTH, M. 1999. Internet-scale push systems for information distribution - architecture, components, and communication. Ph.D. thesis, Technical University of Vienna.

HAUSWIRTH, M. AND JAZAYERI, M. 1999. A component and communication model for push systems. In *Proceedings of Software Engineering - ESEC/FSE'99*. 20–28.

HOLBROOK, H., SINGHAL, S., AND CHERITON, D. 1995. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proc. of ACM SIGCOMM'95*.

HORSTMANN, M. AND KIRTLAND, M. 1997. *DCOM Architecture*. www.microsoft.com/com/tech/DCOM.asp.

HOUSTON, P. 1998. *Building Distributed Applications with Message Queuing Middleware (White Paper)*.

HUANG, Y. AND GARCIA-MOLINA, H. 2001. Publish/subscribe in a mobile enviroment. In *Proceedings of MobiDE*. 27–34.

LEHMAN, T., LAUGHRY, S. M., AND WYCKOFF, P. 1999. Tspaces: The next wave. In *Proc. of Hawaii International Conference on System Sciences (HICSS-32)*.

LEWIS, R. 1999. *Advanced Messaging Applications with MSMQ and MQSeries*. QUE.

LI, K. AND HUDAK, P. 1989. Memory coherence in shared memory systems. *ACM Transactions on Computers Systems 7,* 4 (Nov.), 321–359.

LIN, J. AND PAUL, S. 1996. A reliable multicast transport protocol. In *Proc. of IEEE INFOCOM'96*. 1414–1424.

OMG. 2001. *CORBA Event Service Specification*. OMG.

OMG. 2002a. *The Common Object Request Broker: Core Specification*. OMG.

OMG. 2002b. *CORBA Notification Service Specification*. OMG.

Oracle 2002. *Oracle9i Application Developer's Guide – Advanced Queuing*. Oracle.

PAPADOPOULOS, G. AND ARBAB, F. 1998. Coordination models and languages. In *The Engineering of Large Systems*. Advances in Computers, vol. 46. Academic Press.

PEREIRA, J., FABRET, F., LLIRBAT, F., AND SHASHA, S. 2000. Efficient matching for web-based publish/subscribe systems. In *Proceedings of CoopIS*.

POWELL, D. 1996. Group communication. *Communications of the ACM 39,* 4 (Apr.), 50–97.

RATNASAMY, S., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. Application-level multicast using content-addressable networks. In *Proceedings of the Third International Workshop on Networked Group Communication*.

ROSENBLUM, D. AND WOLF, A. 1997. 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*. 344–360.

ROWSTRON, A. 1998. Wcl: A co-ordination language for geographically distributed agents. *World Wide Web*, 167–179.

SEGALL, B. AND ARNOLD, D. 1997. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG'97)*. http://www.dtsc.edu.au/.

SEGALL, B., ARNOLD, D., BOOT, J., HENDERSON, M., AND PHELPS, T. 2000. Content Based Routing with Elvin4. In *AUUG2K*. Canberra, Australia.

SESSIONS, R. 1997. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons.

SKEEN, D. 1998. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview*. http://www.vitria.com.

Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. 2002. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*.

Sullivan, K. and Notkin, D. 1990. Reconciling environment integration and component independence. In *Proceedings of the fourth ACM SIGSOFT symposium on Software development environments*. ACM Press, 22–33.

Sun. 2000. *Java Remote Method Invocation Specification*.

Sun. 2002. *JavaSpaces Service Specification*.

Tai, S. and Rouvellou, I. 2000. 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)*. 308–330.

Tam, M., Smith, J., and Farber, D. 1990. A taxonomy-based comparison of several distributed shared memory systems. *Operating Systems Review 24,* 3 (July), 40–67.

Tay, B. H. and Ananda, A. L. 1990. A Survey of Remote Procedure Calls. *ACM Operating Systems Review 24,* 3 (July), 68–79.

TIBCO. 1999. *TIB/Rendezvous (White Paper)*.

Wessels, D. 1995. Intelligent caching for world-wide-web objects. In *Proc. of INET'95*. Honolulu, Hawaii.

Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. 1987. *Object-Oriented Concurrent Programming*. MIT Press, Chapter Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, 55–89.

Zhuang, S., Zhao, B., Joseph, A., Katz, R., and Kubiatowicz, J. 2001. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*.

Received January 2001; December 2002; accepted March 2003