1

Introduction

Publish/subscribe (pub/sub) technology encompasses a wide number of solutions that aim at solving a vital problem pertaining to timely *information dissemination and event delivery* from publishers to subscribers [1, 2]. In this chapter, we give an overview to pub/sub systems, examine their history, and motivate the contents and structure of this book.

1.1 Overview

The pub/sub paradigm is very useful in describing and monitoring the world around us. Any person meets a constant barrage of events in his waking hours. Most of these events are irrelevant and they should not be allowed to consume the decision maker's resources of awareness, watchfulness, processing and deciding upon actions. Some events are useful to notice and then there are others which are important, even critically important and create the need to muster all the tools and resources to hand. The ability to be aware of a rich stream of events with minimal exertion and to immediately detect critical events for further processing is central to any successful person or organization. The task of efficient event awareness is formidable.

There are a couple of mitigating factors, though. Typically we might know something about the probable sources of interesting events, although we are not actually interested in knowing who sends the notification of an event. Also we might know in advance something about the type of interesting events and can use this knowledge to preselect sources and also to recognize which are critical events. Thus we are interested in event streams of certain types and sources. One can say that we want to subscribe only such a subset of events streams that is enriched for our purposes.

For digital communication purposes this can be interpreted like this: we need a useful communication paradigm, a pub/sub, also called event notification, service that enables the communication components to dynamically detect and isolate particular events. Simultaneously the pub/sub service must allow introduction of new kinds of events. The participating components are generally unaware of each other, that is, an event may be sourceless from the viewpoint of the receiver.

COPYRIGHTED MATERIAL

Publish/Subscribe Systems: Design and Principles, First Edition. Sasu Tarkoma.
© 2012 John Wiley & Sons, Ltd. Published 2012 by John Wiley & Sons, Ltd.

The pub/sub information dissemination and event delivery problem can be stated as follows: How to deliver information from its publishers to interested and active subscribers in an efficient and timely manner? Information is delivered in the form of asynchronous events, which are first detected, and then delivered by publishers to active subscribers in the form of notification messages.

The problem is vital, because many applications require timely data dissemination. To give some examples, stock market data updates, online advertising, asynchronous events in a *graphical user interface (GUI)*, purchase and delivery tracking, digital news delivery, online games, Web feeds (RSS), and in signalling in many embedded and industrial systems. Indeed, pub/sub is a general enabler for many different kinds of applications and it is especially useful in connecting distributed components together forming a basis for loosely coupled systems.

This problem is also challenging, because the information delivery and processing environments can be diverse and a single technological solution cannot address all these environments and the scenario specific requirements. Thus many different pub/sub systems have been developed. Research oriented systems have demonstrated algorithms, structures, and optimizations to pub/sub technology being applied in a certain operating environment. Industry standards have defined the conventions, interfaces, and *Application Programming Interfaces (APIs)* for creating interoperable pub/sub-based products and solution that use the technology. Thus academic research and industry standardization address two different but partially overlapping facets of the information dissemination problem.

Pub/sub and event-based systems are very different from database systems, because they enable data dissemination from publishers to subscribers in the present and future. This contrasts the traditional database model, in which queries are performed on existing data that is available in a database. The notions of database query and subscription are similar, but the query is about the past whereas the subscription is about the future when it is issued. Data tuples stored in a database and the published event, or notification, are also similar, but differ in that the event is forwarded from the publisher to the subscriber and is not stored by the pub/sub system other than for queuing purposes.

Pub/sub is a broad technology domain and consists of many solutions for different environments. Experiences in building pub/sub solutions and implementing them suggest that no single solution is able to meet the demands of the differing application environments and their requirements. This is evident in the number of pub/sub related standards, implementations, protocols, and algorithms. Yet, the goal of connecting diverse communicating entities through a substrate that supports asynchronous one-to-many communication is shared by these solutions.

Pub/sub is a potential candidate to become a key enabler for Web and mobile applications. On the Web, pub/sub enables the asynchronous communication of various Web components, such as web pages and web sites. Figure 1.1 presents a vision for content dissemination on the Internet that has inspired Google's Pubsubhubbub system.[1] In this vision, anyone can become content publisher and aggregator. Open

[1] http://code.google.com/p/pubsubhubbub/.

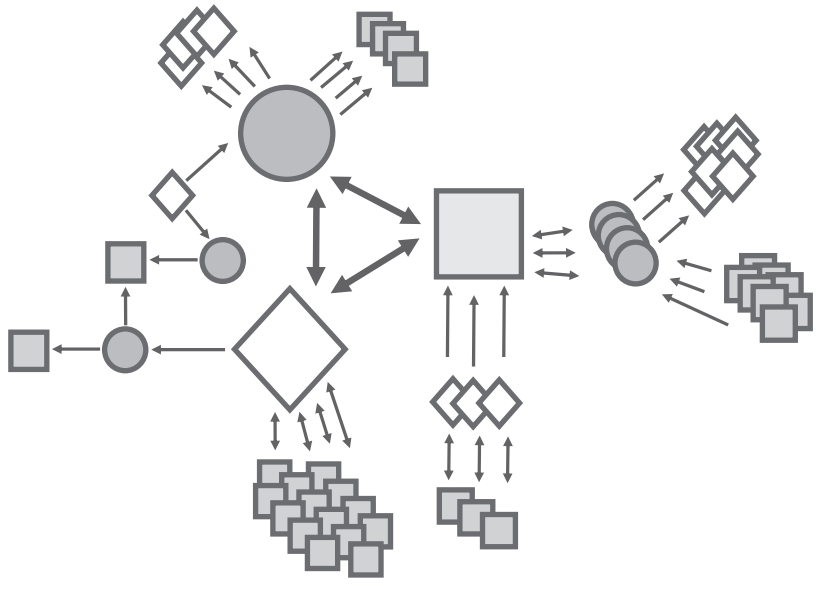

Figure 1.1 A vision of a self-organizing content dissemination system.

interfaces and protocols allow the integration of various content sources. Some publishers and sites become large and others remain small and topical.

Popular alert services, such as Google Alerts[2] and Microsoft Live Alerts[3] allow end users to input keywords and receive relevant dynamic Web content. They are examples of centralized pub/sub solutions for the Web. Their implementation details are not available, but it is believed that alert services are still based on batch processing through search engines. The search engines need to crawl and index live content. Except for a small number of frequently crawled selected sites, the crawling period is typically in the order of a week or tens of days. Thus, they offer a limited form of pub/sub. The next step would be a more decentralized, scalable, and real-time service with support for expressive content matching. Unfortunately, expressive matching semantics and scalability contrast each other making the design, implementation, and deployment of such a global pub/sub service challenging.

Architecture and protocol design should support self-organization and preferential attachment to content sources as well as efficient and timely content dissemination from content publishers through the intermediaries to the content subscribers. The mechanism, techniques, and algorithm are in the key focus of this book. We will address the different facets of the information dissemination problem, and present a collection of frequently employed pub/sub solutions as well as guidelines on how to apply them in practice.

[2] http://www.google.com/alerts.
[3] http://alert.live.com.

1.2 Components of a Pub/Sub System

Before going deeper into the topic, we first define the central terms and components, and the overall structure of a pub/sub system.

1.2.1 Basic System

The main entities in a pub/sub system are the publishers and subscribers of content. A publisher detects an event and then publishes the event in the form of a notification. A notification encapsulates information pertaining to the observed event. The notification can also be called the event message.

There are many terms for the entities in pub/sub or event systems; for example, the terms subscriber, consumer, and event sink are synonymous. Similarly, publisher, producer, supplier, and event source are synonymous. As mentioned above, the notification or event message denotes that an observed event has happened.

An event represents any discrete state transition that has occurred and is signalled from one entity to a number of other entities. For example, a successful login to a service, the firing of detection or monitoring hardware and the detection of a missile in a tactical system are all events.

Events may be categorized by their attributes, such as which physical property they are related to. For instance spatial events and temporal events note physical activity. Moreover, an event may be a combination of these, for example an event that contains both temporal and spatial information. Events can be categorized into taxonomies on their type and complexity. More complex events, called composite or compound events, can be built out of more specific simple events. Composite events are important in many applications. For example, a composite event may be fired

- in a hospital, when the reading of a sensor attached to a patient exceeds a given threshold and a new drug has been administered in a given time interval;
- in a location tracking service, where a set of users are in the same room or near the same location at the same time; or
- in an office building, where a motion detector fires and there has been a certain interval of time after the last security round.

After the notification has been published, it is the duty of the pub/sub system to deliver the message to interested recipients – the subscribers. A subscriber is an entity that has expressed prior interest to a set of events that meet certain requirements that the subscriber has set. The actual delivery depends on the pub/sub solution being used; for example, it could be based on the following:

- The message is broadcast on the network and devices on the same network will see the message. The pub/sub system running on a device can then process the message and deliver it to the subscriber if it is active on the device.
- The message is delivered via network supported multicast, in which a specific network primitive is used for delivering the message from one publisher to many subscribers.
- The message is sent directly by the publisher to subscribers that have informed the publisher that they are interested in receiving a notification. The publisher then utilizes

a one-to-one message delivery protocol on top of the communication primitives offered by the network, typically the TCP/IP protocol stack.
- The message is first sent to a broker server and then delivered by the broker to active subscribers. In this case, the subscribers have expressed their interest in receiving notifications with the broker.
- The message is delivered through a network of brokers. The scalability of a pub/sub system can be increased by deploying a network of pub/sub brokers.

The two first cases are based on communication primitives provided by the underlying network, namely broadcast and multicast. Typically these primitives are not usable with Internet applications, because they are supported only within specific regions of the Internet and thus cannot be used to deliver messages in the global environment. The third case is very typical and extensively used when the number of subscribers is known to be small. This strategy does not scale when the number of subscribers increases. The fourth and fifth case introduce the concept of a broker, also called pub/sub router, that mediates events and provides a routing and matching engine for the publishers and subscribers. This is a commonly used solution for the distributed environment. A well-known technique for deploying pub/sub systems is to create them as overlay networks that operate on top of the current Internet routing system [3].

1.2.2 Distribution and Overlay Networks

A pub/sub system may be centralized or distributed in nature. The notification processing and delivery responsibility may be provided by different entities:

- publishers;
- a centralized broker;
- a set of brokers in a routing configuration typically realized as an overlay network.

Event and notification processing can be easily implemented in publishers and with a centralized broker; however, as observed above, these approaches do not scale well when there are many entities and events in the system. Scalability can be improved by implementing the pub/sub system with a set of brokers as an overlay construct on top of the network layer.

An application layer overlay network is implemented on top of the network layer and it provides services such as resource lookup, overlay multicast, and distributed storage. An overlay network typically provides useful features such as easy deployment of new distributed functions, resilience to network failures, and fault-tolerance [3]. An overlay-routing algorithm is based on the underlying packet-routing primitives. A pub/sub overlay system is implemented as a network of application layer brokers or routers that communicate by using the lower layer primitives, typically TCP/IP.

Figure 1.2 illustrates a pub/sub overlay network. The two important parts of a distributed pub/sub network are the broker topology and how routing state is established and maintained by the brokers. By propagating routing state we mean how the interests of the subscribers are sent towards the publishers of that information. In essence, the routing state stored by a broker must enable it to forward event messages either to other brokers or to subscribers that have previously subscribed to the notifications.

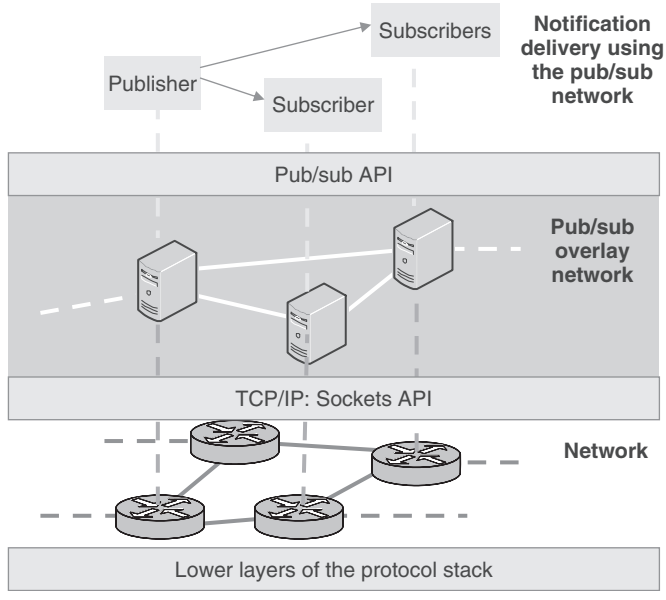

Figure 1.2 Example of a pub/sub overlay network.

In this book, we will investigate the above ways of realizing the notification as well as solutions for achieving high performance, expressiveness, availability, fault resilience, and security.

1.2.3 Agreements

The pub/sub system is used to facilitate the delivery of the messages; however, the meaning of the event is application and domain specific. In order to build a pub/sub system with many entities the following agreements need to be considered:

- Agreement of the notification message format and syntax. For example, many systems utilize a typed-tuple-based format or XML. This agreement may consist of additional details such as those pertaining to timestamps and content security.
- Agreement of the message protocol that is used to transfer the event between two entities. This can include many parameters, for example security, reliability, etc.
- Agreement of the notification filtering semantics. This specifies what elements of the message can be used to make a notification decision. For example, a notification is forwarded based on the publisher, observation time, and type of the event.
- Agreement on the visibility of the published event. It may be necessary to restrict the delivery and processing of the event in the operating environment.
- Agreement of the application and domain specific interpretation of the event. This agreement is outside the scope of a pub/sub system.

Thus many implicit or explicit agreements are needed to design and implement a pub/sub system for an environment that consists of many entities.

1.2.4 The Event Loop

The *event loop* is a key construct in creating event-based applications. The event loop is a frequently used approach in implementing applications that react to various events. For example, Microsoft Windows programs are based on events. The main thread of the application contains the event loop, which waits for new events to process. The event loop can use a blocking function call for receiving messages or a nonblocking peek message function. Typically when a message is received it is processed and delivered to callbacks for further processing.

The event loop is a crucial part of an application that needs to react to events in a timely manner, for example GUI events. The event loop naturally combines with a distributed pub/sub system and it is a key construct for implementing pub/sub engines. A simple pub/sub engine can be implemented as an event loop that reacts to incoming subscription and publishing requests.

1.2.5 Basic Properties

Pub/sub technology has evolved since its inception in the late 1980s to a promising technology for connecting software components across space, time, and synchronization [4]. These three properties summarize the salient features of the technology. We will examine each of the three properties presented in Figure 1.3 in detail in this section.

Space decoupling is illustrated by subfigure A, in which the event notification service decouples the publisher and the subscribers. The event message is transferred to the event service, and then it is transferred to the subscribers. Thus memory space is not shared by the entities. Subfigure B presents an example of time decoupling. The setting is the same as for the space decoupling case with the exception of message buffering at the service side. Time decoupling is achieved by storing the message in a message buffer at the event notification service for eventual delivery to subscribers. The synchronization decoupling is illustrated by subfigure C, which emphasizes the temporal aspect. The publish and notify phases of event delivery are decoupled and they do not require synchronization. The message is first delivered to the event notification service and then to the subscriber.

Figure 1.4 summarizes the decoupling properties of well-known communication techniques. As observed before, the communication techniques are not orthogonal but rather they are combined in order to implement more sophisticated systems. Message passing, *Remote Procedure calls (RPC)* and *Remote Method Invocation (RMI)*, and asynchronous RPC/RMI do not offer decoupling in space and time. They can offer decoupling in synchronization. Tuple spaces offer decoupling in space/time through the shared space; however, the reader of the tuple space is blocked and thus tuple spaces do not offer decoupling in synchronization [5]. Message queuing, on the other hand, offers decoupling of all three properties and it is a building block for the more sophisticated pub/sub systems.

Pub/sub is based on message queuing and message-oriented middleware. Message queuing is a communication method that employs message passing between a sender and a receiver with the help of a sender-side message queue. A message being sent is first stored in the local message queue. After the delivery has been made, the message can be removed from the queue. If the message cannot be delivered or the message is incorrectly received, the message can be resent.

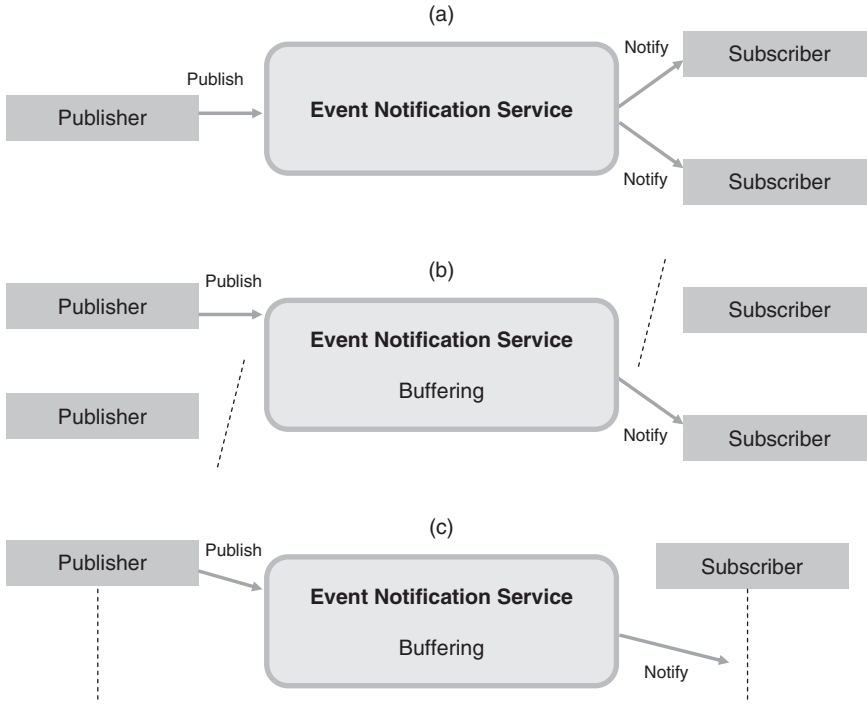

Figure 1.3 Decoupling properties in pub/sub. (a) Space decoupling; (b) Time decoupling; (c) Syschronization decoupling.

Abstraction	Space/time decoupling	Synchronization decoupling
Message passing	No	Varies
RPC/RMI	No	Invoker is blocked
Async RPC/RMI	No	Yes
Tuple spaces	Yes	Reader is blocked
Message queuing	Yes	Varies
Pub/sub	Yes	Yes

Figure 1.4 Summary of decoupling properties.

Queuing is a basic solution in achieving reliability in data communications. Queuing also supports disconnections during which the message cannot be sent. Message queuing is thus the basic ingredient for achieving decoupled communications.

One distinction between message queuing systems and pub/sub is that they typically offer one-to-one communications and require that the receivers are explicitly defined. Pub/sub on the other hand supports one-to-many and many-to-many communications and the subscribers can be defined implicitly by the event message being delivered and the a priori subscriptions that the subscribers have set.

The key properties of pub/sub systems are: decoupling in space, time and synchronization, many-to-many communications, and information filtering.

1.3 A Pub/Sub Service Model

Figure 1.5 illustrates a generic pub/sub service design. In the figure, the pub/sub service is a logically centralized service that provides the necessary functions and interfaces for supporting notification delivery from publishers to subscribers. The pub/sub service consists of the following key components:

- A notification engine that builds and maintain an index structure of the subscriptions, and uses the index table to forward notifications to subscribers. The engine offers the necessary interfaces for subscribers and publishers that allow them to subscribe, unsubscribe, and publish content.
- A subscription manager that accepts subscriptions from the engine and maintains those. The two mandatory operations are insert and remove a subscription.
- A subscription storage that stores subscriptions and data related to the subscriptions.
- An event storage is a facility for storing published events so that they can be retrieved later.
- A notification consumer that is an intermediary component in the notification process. A consumer receives notifications from the engine and then forwards those to the proper subscriber. The consumer can buffer, compress, and process notifications before the final delivery.

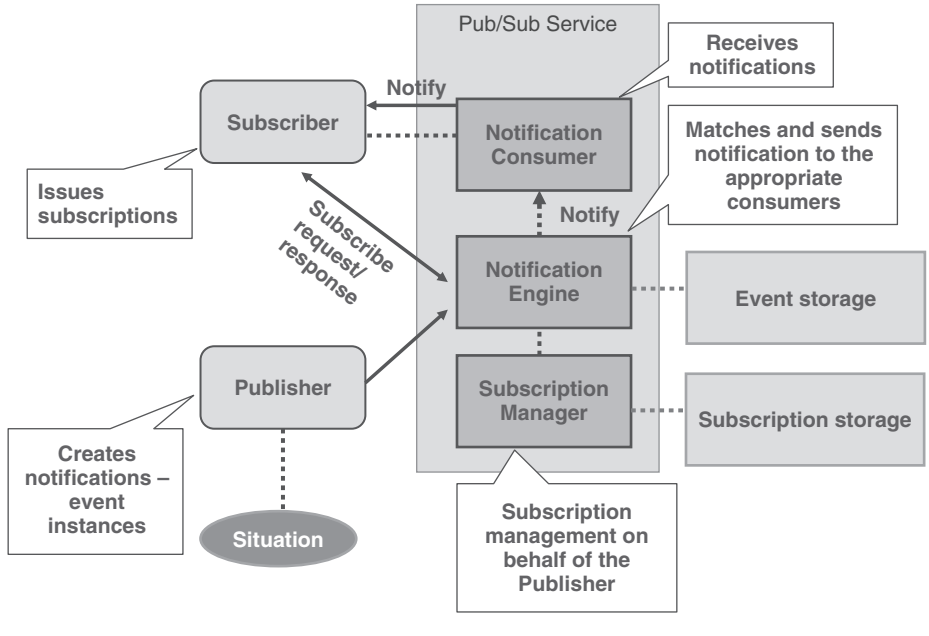

Figure 1.5 Components of a pub/sub system.

A publisher observes a situation and when an event of interest is observed, a notification is created and sent to the notification engine using its publication interface. The notification is then matched with the subscription index maintained by the engine with the help of the subscription manager. The notification is given by the engine to the notification consumers of subscribers that have expressed interest in the notification. In other words, the notification matches with the subscribers' subscriptions. The notification is then prepared by each consumer for delivery to the associated subscriber.

This model of a pub/sub service separates the management of the subscriptions, the matching process with the notification engine, and the final delivery to the subscribers. This separation allows, for example, changing of the notification consumer without changing the engine.

The design of Figure 1.5 is logically centralized and it hides the distribution of the components. It is necessary to distribute and replicate the components in order to achieve scalability and reliability in a distributed environment.

1.4 Distributed Pub/Sub

As mentioned in this chapter, direct notification of subscribers by a publisher is not scalable. Therefore it is vital to develop techniques for distributing the notification process. To this end, a number of pub/sub network designs have been developed.

An *event broker or router* is a component of a pub/sub network that forwards notification messages across multiple hops. An example pub/sub network is presented in Figure 1.2 that shows the layered design. The pub/sub network offers the notification API to subscribers and publishers and utilizes the network API, typically the Sockets API, to disseminate the notification message and take it from the source router to the destination router and subnetwork. The network level routers are responsible for taking the message end-to-end across the Internet. Such overlay designs have favourable characteristics in terms of deployability and flexibility; however, the resulting high level routing may not be efficient in terms of the network level topology.

An event router typically has local clients and neighbouring routers. The algorithms and protocols for local clients and neighbouring routers are different. Both cases require a *routing table* for storing information about message destinations. A pub/sub routing table is an index structure that contains the active subscriptions and typically supports add, remove, and match operations.

The design and configuration of pub/sub networks has become an active area of research and development. We will focus on various strategies for implementing pub/sub networks.

The simplest form of notification in the distributed environment is called *flooding*. With flooding each pub/sub broker simply sends the message to all neighbours except the one that sent the message. Thus the message is introduced at every broker; however, the price of the technique is its inaccuracy. Ideally, we want to prevent the forwarding of a message to a broker that we know does not have subscribers for the message. Moreover, excess and uncontrolled messaging may lead to congestion that in turn may cause notification messages to be dropped.

In order to avoid unnecessary message deliveries, we introduce the notion of *filtering* into the pub/sub network. Filtering involves an interest registration service that accepts filter information from the subscribers. The subscribers can thus specify in more detail

what kind of data they desire. The pub/sub network then distributes this filtering information in such a way that minimizes the overhead in notification message delivery. The process of optimizing a pub/sub network is not simple, because the filtering information also introduces overhead into the network. For example, filtering information may need to be updated, and there is propagation delay in setting up and maintaining the routing tables of pub/sub brokers. Later in this book we will consider various techniques in optimizing these networks.

Accuracy is a key requirement for a pub/sub network. The accuracy of event delivery can be expressed with the number of false positives and false negatives.

A *false positive* is a message that is sent to a subscriber that does not match the subscriber's active interests. Similarly, a *false negative* is a message that was not sent to a subscriber, but should have been because it matches the subscriber's active interests.

Various filtering languages and filter matching algorithms have been developed. Filtering involves the specification of filters that are organized into a filtering data structure. A filter selects a subset of notifications based on the filtering language. Thus a filter is a constraint on the notification message and it can be applied in the context of the notification type, structure, header, and content.

Filtering allows the subscribers to specify their interest beforehand and thus reduce the number of uninteresting event messages that they will receive. A filter or a set of filters that describes the desired content is included with the subscription message that is used by brokers to configure routing tables. Many filtering languages have been developed, specified, and proposed. For example, the filtering language used by *Java Message Service (JMS)* is based on *Structured Query Language (SQL)* [6].

Filtering is a central core functionality for realizing event-based systems and accurate content-delivery. Filtering is performed before delivering a notification to a client or neighbouring router to ensure that the notification matches an active subscription from the client or neighbour. Filtering is therefore essential in maintaining accurate event notification delivery.

Filtering increases the efficiency of the pub/sub network by avoiding to forward notifications to brokers that have no active subscriptions for them. Filters and their properties are useful for many different operations, such as matching, optimizing routing, load balancing, and access control. To give some examples, a firewall is a filtering router and an auditing gateway is a router that records traffic that matches a given set of filters.

1.5 Interfaces and Operations

Table 1.1 presents the pub/sub operations used by many event systems [7]. The operations are requested by a client, denoted by X, of the system. There are many ways to define the interests of the subscriber. In our generic API, we denote the general interests by F.

Table 1.1 Infrastructure interface operations

Operation	Description	Semantics
Sub(X,F)	*X* subscribes content defined by *F*	Sub/Adv
Pub(X,n)	*X* publishes notification *n*	Sub/Adv
Notify(X,n)	*X* is notified about notification *n*	Sub/Adv
Unsub(X,F)	*X* unsubscribes content defined by *F*	Sub/Adv
Adv(X,C)	*X* advertises content *C*	Adv
Unadv(X,C)	*X* unadvertises content *C*	Adv
Fetch(X,P)	*X* fetches messages that satisfy the given constraints *P*	Sub/Adv

In expressive content-based routing F is typically defined with a Boolean function that selects a subspace of the content space, in which the notifications are defined. Notifications are points in this space. There are also less expressive semantics for subscribing content, such as type-based subscriptions. We will return to these notions shortly.

As presented by the table, the key operations pertain to publishing, subscribing, unsubscribing, and fetching content. It should be noted that the subscribe and unsubscribe operations are idempotent, which means that even if the same operation is executed repeatedly it does not change the state of the system. Publish operation, however, is not idempotent and repetitions will cause many publications to be delivered.

In a large-scale pub/sub system, the API typically supports leases that determine the validity time period for each subscription and advertisement. Leases are useful in removing obsolete state from the pub/sub network, and they are instrumental in ensuring the eventual stability of the network. The unsubscription and unadvertisement are not necessary if leases are supported by the API; however, they may still be useful in terminating a lease before the it expires.

There are two different kinds of operational semantics for a pub/sub system:

- Subscription-driven: Subscriptions are propagated by the pub/sub network and the routing tables are based on filters specified in the subscription messages.
- Advertisement-driven: Publishers first advertise content with advertisement messages that are propagated by the pub/sub network. The pub/sub network then connects subscriptions with matching advertisements to active content delivery across the network.

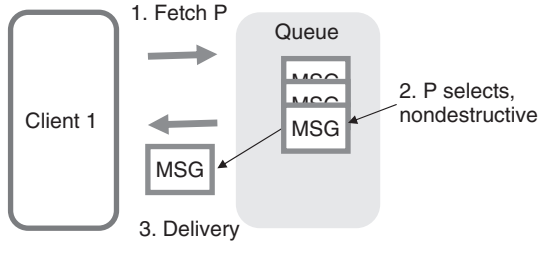

Figure 1.6 Example of the expressive fetch operation.

The table presents the API operations for these two filtering semantics. The advertisement semantics introduces the operations for advertising and unadvertising content. Moreover, the API operations are typically extended with security and quality-of-service properties as well as more expressive notification retrieval strategies. Key extensions pertaining to event retrieval, the fetch operation illustrated by Figure 1.6, include:

- Fetch operation for retrieving a specific number of messages.
- Nondestructive fetch, which leaves the retrieved messages at the server's queue. This is useful when multiple instances of the same software are retrieving messages.
- Fetch operation with query operation that allows specific event messages to be fetched from the queue. This operation is frequently supported by pub/sub standards, for example JMS.
- Fetch the latest event message in the message queue. This is useful when starting an application or recovering from application failure.

In the following section we will investigate the different filtering semantics for targeted information delivery.

1.6 Pub/Sub Semantics for Targeted Delivery

As mentioned above, there needs to be agreement on how notification messages are delivered from publishers to subscribers. There are many possible semantics for selecting notifications that need to be delivered for a given set of subscribers. In this section, we will briefly examine key semantics for targeted notification delivery.

Depending on the expressiveness of the filtering language, a field, header, or the whole content of the notification message may be filterable. In *content-based routing* the whole content of the event message is filterable.

Figure 1.7 illustrates the four key types of message routing semantics. The types are the following: content-based, header-based, topic-based, and type-based. As mentioned above, content-based routing allows the evaluation of filters on the whole event message. Header-based is more limited and only allows to evaluate elements included in the header of the message. Topic-based only allows to evaluate a specific topic field in the message. Topic-based systems are similar to channel-based systems and the topic name can be seen to be the same as the channel name. Typically topic-based systems require that the topic of an event message exactly matches with the requested topic name and thus it is not very expressive. Finally, type-based systems allow the selection of event messages based on their designated type in a type hierarchy. We can take a type hierarchy pertaining to buildings as an example: the root of the hierarchy is the building name, the second level consists of floors, and the third level of the offices. By subscribing to a floor the subscriber receives all events related to that specific floor in the named building.

The different routing semantics are characterized by their selectivity. Type-based systems make the forwarding decision based on a predefined set of message types. In topic-based and channel-based pub/sub, the subscribers are defined by a queue name or a channel name. The notifications are sent to a named queue or channel, from which the subscriber then extracts the messages. An important limitation is that the queue or channel name has to be agreed beforehand. Subject-based systems make the routing decision based

Name	Value
Resource_name	CS Department's home page
Address	www.cs.helsinki.fi
Resource_type	Web page
Content element 1	Data

Content-based routing

Name	Value
Resource_ name	CS Department's home page
Address	www.cs.helsinki.fi
PAYLOAD	

Header-based routing

Name	Value
Topic	CS Department Channel
PAYLOAD	

Topic-based routing

Name	Value
Type	UH\Faculty of Science\CS Department
PAYLOAD	

Type-based routing

Figure 1.7 Examples of message targeting systems.

on a single header field in the notification. Header-based systems use a special header part of the notification in order to forward the message. Content-based systems are the most expressive and use the whole content of the message in making the forwarding decision. Content-based pub/sub is flexible because it does not require that topic or channel names are assigned beforehand.

> Various pub/sub delivery semantics can be implemented with a content-based communication scheme making it very expressive. Header-based routing is more limited, but it has a performance advantage to content-based routing, because only the header of a message is evaluated when making a forwarding decision.

Expressiveness and scalability are important characteristics of an event system [8]. Expressiveness pertains to how well the interests of the subscribers are captured by the pub/sub service. Scalability involves federation, state, and the number of subscribers, publishers, and brokers can be supported as well as the how much notification traffic can the system support.

Other requirements for a pub/sub network include simplicity, manageability, implementability, and support for rapid deployment. Moreover, the system needs to be extensible and interoperable. Other nonfunctional requirements include: timely delivery of notifications (bounded delivery time), support for *Quality of Service (QoS)*, high availability and fault-tolerance.

Event order is an important nonfunctional requirement and many applications require support for either causal order or total order. Causality determines the relationship of two events A and B. In order to be able to determine causality in the distributed system a logical clock mechanism is needed. The two well-known solutions are the Lamport clocks and vector clocks. We will examine these clocks in more detail in Chapter 2.

1.7 Communication Techniques

Event systems are widely used, because asynchronous messaging provides a flexible alternative to RPC [4, 9]. RPC is typically synchronous and one-to-one, whereas pub/sub is asynchronous and many-to-many. Limitations of synchronous RPC calls include:

- Tight coupling of client and server lifetimes. The server must be available to process a request. If a request fails the client receives an exception.
- Synchronous communication. A client must wait until the server finishes processing and returns the results. The client must be connected for the duration of the invocation.
- Point-to-point communication. Invocation is typically targeted at a single object on a particular server.

On the other hand, RPC is a building block for distributed pub/sub systems. Many pub/sub implementations use RPC operations to implement the API operations presented in Table 1.1.

Event delivery between two processes can be realized in many ways depending on the requirements and the operating environment. The two key differing environments are the local and remote communication context. In local event delivery, techniques such as shared resources and local procedure calls or message passing can be used. Remote event delivery is typically implemented with message queuing or RPC.

RPC offers reliability and at-most-once semantics whereas message queuing systems have differing message delivery options. The key reliability semantics are:

- Exactly-once: The highest reliability guarantee in which the message is sent to the remote node exactly once. The message is delivered even if the server crashes or the network fails. This reliability level is not possible to achieve in the typical distributed environment.
- At-least-once: This reliability level guarantees that a message is sent to the remote node, but duplicates are allowed. Duplicates may happen due to network failures and server crashes. The semantics are appropriate for idempotent operations.
- At-most-once: This reliability level guarantees that the message is sent to node once if at all. It does not guarantee that message is delivered. A message can disappear due to network problems or server crashes.

Typically commercially used message queue systems support either at-least-once or at-most-once. The semantics are implemented with sender and receiver side message buffering, sequence numbers, and timers for detecting lost messages and other problems.

Figure 1.8 illustrates the options when implementing pub/sub systems. The key difference between message queuing and RPC is that messaging is asynchronous whereas traditional RPC is synchronous although there are also asynchronous RPC features. Alternative techniques are tuple spaces and distributed shared memory. We will later in Chapter 2 consider Java RMI as one example of an RPC system.

Distributed shared memory can be realized in many ways based on the memory abstraction. A page based abstraction organizes the shared memory into pages of fixed size. The object based abstraction organizes the shared memory as an abstract space for storing shareable objects. A tuple space, on the other hand, is based on the

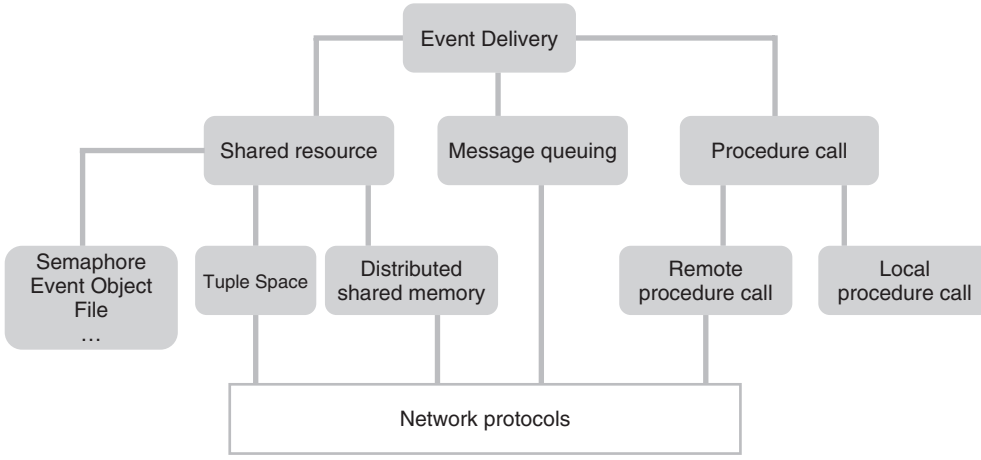

Figure 1.8 Communication techniques for event delivery.

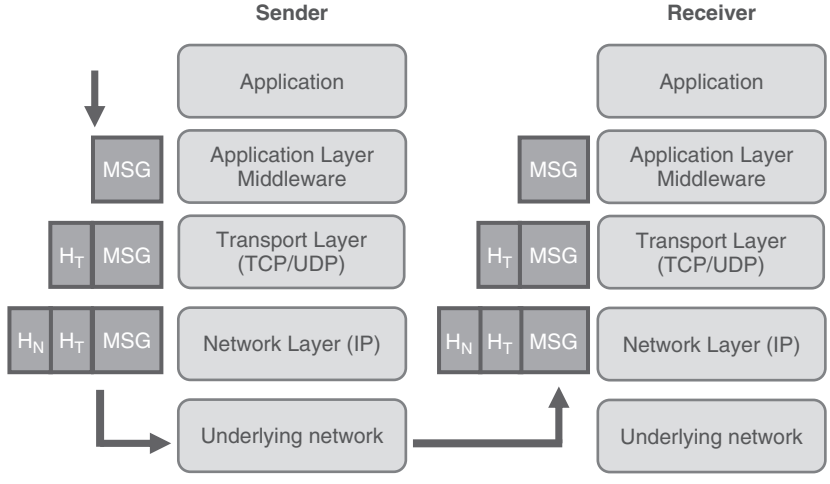

Figure 1.9 Protocol stack with middleware.

tuple abstraction. A coherence protocol is needed to maintain memory coherence in a distributed shared memory implementation. Memory update and invalidation techniques include update-on-write, update-on-read, invalidate-on-write, invalidate-on-read. Typically these systems follow the weak consistency model, in which synchronizations must be atomic and have a consistent order.

Figure 1.9 illustrates the layered nature of the communications environment. Each layer provides functions for the higher layers and abstracts details of the underlying layers. The organization of protocols into a stack structure offers separation of concerns; however, it makes it difficult to optimize system behaviour across layers. As shown by the figure, each layer adds its own header and details to the packets and messages being processed.

In a similar fashion, when receiving a packet, each layer processes its own information and gives the data to a higher layer. Pub/sub systems can be implemented on multiple levels of the stack, starting from the link layer towards the application layer. Most pub/sub systems are implemented on top of TCP/IP and they are offered as middleware services or libraries. A pub/sub system can itself be viewed to be a layered system, in which the higher level functions of distributed routing and forward are based on lower layer message queuing primitives.

1.8 Environments

The pub/sub paradigm can be applied in many different contexts and environments. Early examples include GUIs, control plane signalling in industrial systems, and topic-based document dissemination. The paradigm is fundamental to todays's graphical and network applications. Most programmers apply the paradigm in the context of a single server or device; however, distributed pub/sub is vital for many applications that require the timely and efficient dissemination of data from one or more sources to many subscribers.

The operating environments for pub/sub can be examined from differing viewpoints, for example based on the underlying communications environment and the application type. In the following we summarize key environments for pub/sub technology:

- Local: Event loop, GUI, in-device information delivery.
- Wireless and ad hoc: Event delivery in wireless networks in which nodes can move. Publishers and subscribers are typically run on constrained limited devices, for example mobile phones.
- Sensor: Event delivery in sensor networks from a number of source sensors to sinks that then deliver the events for further processing.
- Embedded and industrial: Event delivery in an embedded or industrial setting, for example within a car or an airplane or a factory.
- Regional: Event delivery within an organization or a region.
- Internet-wide: Event delivery in the wide-area network across organizational boundaries.

In this book we will focus especially on distributed pub/sub systems for the last three categories; however, we do also consider the mobile and wireless domain as well.

Small and wireless devices have limited capabilities compared to desktop systems: their memory, performance, battery life, and connectivity are limited and constrained. The requirements of mobile computing need to be taken into account when designing an event framework that integrates with mobile devices.

From the small device point of view, message queuing is a frequently used communication method because it supports disconnected operation. When a client is disconnected, messages are inserted into a queue, and when a client reconnects the messages are sent. The distinction between popular message-queue-based middleware and notification systems is that message-queue-based approaches are a form of directed communication, where the producers explicitly define the recipients. The recipients may be defined by the queue name or a channel name, and the messages are inserted into a named queue, from which the recipient extracts messages. Notification-based systems extend this model by

adding an entity, the event service or event dispatcher, that brokers notifications between producers of information and subscribers of information. This undirected communication supported by the notification model is based on message passing and retains the benefits of message queuing. In undirected communication the publisher does not necessarily know which parties receive the notification.

The pub/sub paradigm and technology can be seen to be a unifying technology that combines the different environments and domains through event delivery. Indeed, pub/sub has been proposed as the new basis for an internetworking architecture; however, there are still many unsolved challenges in applying the paradigm on the global Internet scale. We consider these solutions in Chapter 13.

1.9 History

As mentioned above, pub/sub can applied in many different environments for solving the information dissemination problem. The early applications include the filtering and delivery of Usenet postings as well as being the glue of many GUIs. More recent applications include Internet technologies such as RSS and XMPP as well as the many standards such as JMS, CORBA Notification Service [10], and OMG DDS.

In this section, we will examine the history of pub/sub systems from three viewpoints, namely the research highlights, standardization, and Internet technology. The last category illustrates the importance and applicability of pub/sub for Internet-based applications. Figure 1.10 gives a timeline of the evolution of pub/sub technology for the three categories. In the following, we briefly examine the key developments. We will return to many of these later in the book.

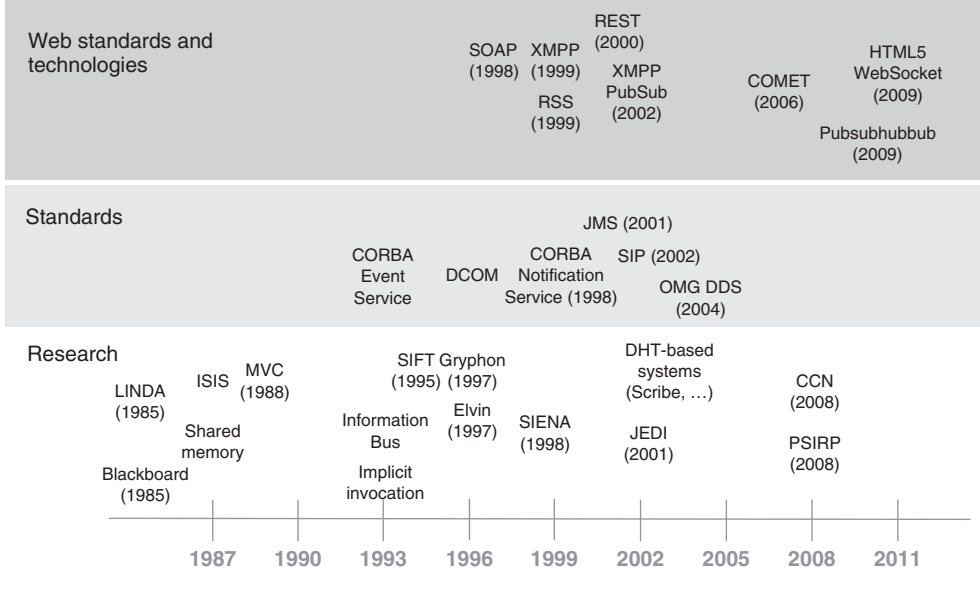

Figure 1.10 Timeline of pub/sub solutions.

1.9.1 Research Systems

The history of pub/sub has its roots in the requirement to process asynchronous events. The notion of the event loop is very old; however, the patterns employed today to realize distributed pub/sub are considerably newer. The first systems were based on the shared memory abstraction. The memory represented rendezvous space for senders and receivers. Processors communicated by posting messages to the shared memory.

The shared memory is very similar to the blackboard pattern frequently used in creating artificial intelligence systems. The blackboard pattern that was proposed in 1985 for solving complex problems with the help of a shared memory abstraction [11, 12]. The LINDA tuple space model is also from this period [5]. LINDA is a coordination and synchronization technique based on the tuple abstraction. LINDA supports communication through a shared memory region called the tuple space. Processes generate tuples and store those in the shared space. Other processes can then monitor the tuple space and read tuples.

Another early example of interprocess communication is the UNIX signal notification system that was implemented in 1986. UNIX processes use signals to notify each other. A process has a unique numeric process identifier and a group of processes have a numeric group identifier. A signal can be directed to a specific process or a group of processes.

The frequently used *Model-View-Control(MVC)* design pattern was developed in the SmallTalk community in 1988. MVC facilitates the communication between the model, view, and control components [13]. The MVC pattern separates concerns over application state (model), user interface (view), and the control aspect. MVC requires that a component is able to subscribe to the state of another component. This subpattern used in the MVC developed into the observer pattern that is widely used. We examine these and other patterns in more detail in Chapter 4.

An early pub/sub service was proposed in the seminal ISIS system in 1987 [14]. This ISIS subsystem was responsible for disseminating news items from publishers to subscribers. The ISIS news service allowed processes to subscribe to system-wide news announcements. The subscriber specified a subject and then received postings under that subject. The ISIS architecture also featured filters that were used on client systems to process incoming messages. The subscribe operation of the news subsystem was implemented with one RPC per posting and the actual posting delivery with one asynchronous multicast operation (with causal or total ordering).

The key contributions of the ISIS system for pub/sub were:

- Reliable atomic multicast communications primitives.
- Causal and total ordering of multicast messages.
- Developing the pub/sub system based on the RPC and multicast primitives.

Another early example of a pub/sub system is the Information Bus proposed in 1993 [15]. This model consists of service objects and data objects. Service objects can have local data objects, and they send and receive them through the datacentric information bus. Each data object is labelled with a subject string. Subjects are hierarchically structured. The Information Bus supports both pub/sub and request/reply APIs. With the pub/sub model, the system decouples the component and subscribers do not need to know the identities of the publishers. The Information Bus model calls this kind of communication subject-based addressing. The system has built-in support

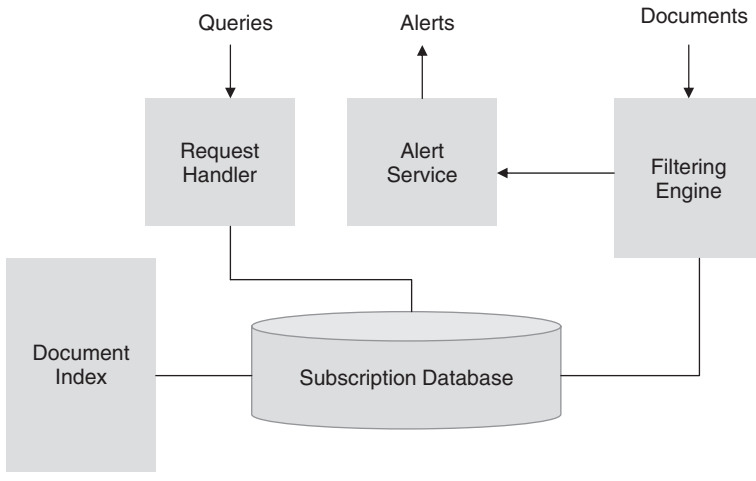

Figure 1.11 Overview of the SIFT system.

for dynamic discovery of participants. This is implemented with two publications, first a query to prospective participants listening a specific subject, and then response publication indicating presence. The system can be extended with adapters that convert information from the data objects to application specific formats.

The ISIS system and Information Bus did not take the content of the messages into account. In the news applications implemented with these systems, the news items were delivered based on the subject that was a configuration parameter. The SIFT information dissemination system is an early example of an alert service [16] that takes the content of the disseminated documents into account. The system proposed the inverted index for matching documents to subscriptions. Figure 1.11 illustrates this system and its key components.

The key idea of the *inverted index* is to allow fast full text searches of the documents. The words need to be extracted during the insertion of the document. Thus the technique places most of the processing cost to the insertion phase. The index structure maps document content, typically the words, into locations in a set of documents. Thus given a query it is easy to determine the matching documents with the inverted index. For example, consider the set of three words {“pub”,“sub”,“event”} with “pub” mapping to documents {1,2}, “sub” to documents {1,3,4}, and event to {2,6,7}. Now, a query for “pub” and “event” would result in $\{1,2\} \cap \{2,6,7\} = \{2\}$. Thus document number two would be returned.

The system accepts document queries and stores them into a subscription database. Similarly, documents are parsed and an inverted index is stored in the document index. A filtering engine then is responsible for matching documents to the queries with the document index.

The SIFT system did not consider the distributed environment in more detail. IBM’s Gryphon project developed a distributed pub/sub system consisting of a network of brokers [17, 18]. The Gryphon system was developed at the Distributed Messaging Systems group at the IBM T.J.Watson Research Center. Gryphon is a Java-based pub/sub message broker intended to distribute data in real time over a large public network. Gryphon

uses content-based routing algorithms developed at the research center. The clients of Gryphon use an implementation of the JMS API to send and receive messages. The Gryphon project was started in 1997 to develop the next generation web applications and the first deployments were made in 1999. Gryphon is designed to be scalable, and it was used to deliver information about the Tennis Australian Open to 50000 concurrently connected clients. Gryphon has also been deployed over the Internet for other real-time sports score distribution, for example the Tennis US Open, Ryder Cup, and monitoring and statistics reporting at the Sydney Olympics.

Gryphon supports both topic-based and content-based publish-subscribe, relies on adopted standards such as TCP/IP and HTTP, and supports recovery from server failures and security. In Gryphon, the flow of streams of events is described using an *information flow graph (IFG)*, which specifies the selective delivery of events, the transformation of events, and the creation of derived events as a function of states computed from event histories.

Elvin is another early example of a content-based routing system with expressive semantics [19]. Elvin uses a client-server architecture in notification delivery. Clients establish sessions with Elvin servers and subscribe and publish notifications.

Scalable Internet Event Notification Service (SIENA) is an Internet-scale event notification service developed at the University of Colorado. SIENA balances expressiveness with scalability and explores content-based routing in a wide-area network. The basic pub/sub mechanism is extended with advertisements that are used to optimize the routing of subscriptions [8]. Several network topologies are supported in the architecture, including hierarchical, acyclic peer-to-peer, and general peer-to-peer topologies. Servers only know about their neighbours, which helps in minimizes routing table management overhead. Servers employ a server-server protocol to communicate with their peers and a client-server protocol to communicate with the clients that subscribe to notifications. It is also possible to create hybrid network topologies.

SIENA introduced covering relations between filters to prevent unnecessary signalling. The SIENA system used the notion of covering for three different comparisons:

- matching a notification against a filter;
- covering relation between two subscription filters;
- and overlapping between an advertisement filter and a subscription filter.

Covering relations have been used in many later event systems, such as Scribe [20], Rebeca [21], and Hermes [22, 23]. Scribe and Hermes are examples of *Distributed Hash Table (DHT)*-based pub/sub systems. Scribe is a topic-based system and Hermes supports both topic-based and content-based communication. Scribe and Hermes choose a rendezvous point for each topic or event type in the overlay network topology, and then build and maintain multicast trees rooted at this rendezvous point. We will later return to DHT structures and DHT-based pub/sub systems.

DHTs are a class of decentralized distributed algorithms. They provide a hashtable API and implement the hashtable functionality in a wide-area environment in which nodes can join and leave the network. A DHT maintains (key, value) pairs and allows a client to retrieve a value corresponding to the given key.

The *combined broadcast and content-based (CBCB)* routing scheme extends the SIENA routing protocols by combining higher-level routing using covering relations and lower-level broadcast delivery [24]. The protocol prunes the broadcast distribution paths using higher-level information exchanged by routers.

Java Event-based Distributed Infrastructure (JEDI) [25] is a distributed event system developed at Politecnico di Milano. In JEDI the distributed architecture consists of a set of *dispatching servers (DS)* that are connected in a tree structure. Each DS is located on a node of the tree and all nodes except the root node are connected to one parent DS. Each node has zero or more descendants.

Gryphon, Elvin, SIENA, and JEDI paved way for the next generation of content-based pub/sub systems developed as overlay networks over the Internet. More recent developments have considered also the introduction of pub/sub primitives into the protocol stack design.

SIENA pioneered the notion of content-based networking, in which content demand defines subnetworks and where information is sent. The notion of datacentric networking is similar and has been pioneered by projects such as TRIAD [26] and DONA [27]. These new forms of networking are motivated by the observation that the current Internet architecture has been designed around a host-based model that dates from the 1970s. The aim is to allow the network to adapt to the network usage patterns and improve performance with targeted information delivery and caching.

For example, the *Publish-Subscribe Internet Routing Paradigm (PSIRP)* system [28] and the *Content Centric Networking (CCN)* architecture [29] are based on receiver driven designs. The motivation is that Internet hosts are interested in receiving the proper content rather than who is supplying the content.

1.9.2 Standards

The standards timeline includes systems such as CORBA Event Service, Microsoft's DCOM, CORBA Notification Service, JMS, SIP, and DDS. In this section we briefly examine these developments.

The CORBA Event Service specification defined a communication model that allowed an object to accept registrations and send events to a number of receiver objects. The Event Service supplements the standard CORBA client-server communication model and is part of the CORBAServices that provide system level services for object-based systems. The CORBA Notification Service [10] extends the functionality and interfaces of the older Event Service [30] specification. The Event Service specification defines the event channel object that provides interfaces for interest registration and event notification. One of the most significant additions to the Notification Service is event filtering.

The *Distributed Component Object Model (DCOM)* was the Microsoft alternative to CORBA technology. DCOM facilitates communication between distributed software components. DOM extends the COM model and provides the communication with COM+ application infrastructure. Today DCOM has been replaced with Microsoft .NET Framework.

Standard COM and OLE supported asynchronous communication and the passing of events using callbacks, however, these approaches had their problems. Standard COM publishers and subscribers were tightly coupled. The subscriber knows the mechanism

for connecting to the publisher (interfaces exposed by the container). This approach does not work very well beyond a single desktop. The change in the COM+ Event Service was the addition of the event service in the middle of the communication. The event service keeps track of which subscribers want to receive the calls, and mediates the calls.

JMS defines a generic and standard API for the implementation of message-oriented middleware. The JMS API is an integral part of the *Java Enterprise Edition (Java EE)*. JMS is an interface and the specification does not provide any concrete implementation of a messaging engine. The fact that JMS does not define the messaging engine or the message transport gives rise to many possible implementations and ways to configure JMS.

The *Session Initiation Protocol (SIP)* [31] is a text-based, application-layer protocol that can be used to setup, maintain, and terminate calls between two or more end points. SIP is designed to be independent of the underlying transport layer. SIP has been designed for call control tasks and thus the driving application has been telephony and multiparty communications. SIP has been standardized by IETF and adopted widely in the telecommunications industry. SIP was accepted as a 3GPP signalling protocol in November 2000.

The *Data Distribution Service for Real-Time Systems (DDS)* OMG specification defines an API for datacentric pub/sub communication for distributed real-time systems [32]. DDS is a middleware service that provides a global data space that is accessible to all interested applications.

1.9.3 Internet Technology

We briefly consider developments on the Internet technology timeline focusing on Web technologies for building pub/sub systems. One of the earliest examples of a loosely coupled message dissemination system is the Usenet that was created in 1980. Usenet thus precedes many other message dissemination systems. Later developments include W3C's SOAP protocol, the *Really Simple Syndication (RSS)* specifications, the *Extensible Messaging and Presence Protocol (XMPP)*, the *Representational State Transfer (REST)* model, HTML5 from W3C, and the Pubsubhubbub protocol. There are also many other systems that have been proposed and deployed.

RSS is a family of specifications for the definition of Web-based information feeds using XML. RSS is a simple pub/sub system that is based on polling the URL that identifies a feed and then determining if information has changed. RSS builds on existing Web standards, namely HTTP and XML, and it has become ubiquitous. RSS is used to disseminate updates, for example, pertaining to blog entries, news, video and audio resources.

SOAP is a key messaging protocol for Web applications that was designed for XML-based RPC and messaging. SOAP is a one-way message exchange primitive specified and standardized by W3C that is very flexible. SOAP can support various interactions by building on the one-way message exchange primitive and can be used with various message transport protocols, such as HTTP and SMTP.

XMPP [33] (RFC 3920 based on the Jabber protocol) has been designed for instant messaging with support for extensions. Today XMPP can support different message-based communication styles. XMPP extensions include publish/subscribe mechanisms, presence and status updates, alerts, feature negotiation, service discovery, and other features that make it suitable as an asynchronous middleware solution. XMPP is becoming increasingly

popular on the Internet with companies such as Google, Twitter,[4] and Facebook[5] using XMPP as a general API.

REST was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. In this model, clients send requests to servers, servers process requests and return responses. The key idea is that the requests and responses convey representations of resource state. A resource is an entity that can be addressed. The model maps well to the HTTP protocol and has influenced the design of Web application interfaces. For instance, the popular real-time feed service Twitter has a REST API.

Comet is an *AJAX (Asynchronous Javascript and XML)* implementation of a push system over the Web. Comet uses callback functions to handle responses from a server. Comet issues HTTP requests to keep the connection to the server open. A long-lived connection is established that is then used to send and receive event data.

HTML5 is a core new language for defining content for the Web. The specification is still under development at W3C and when completed it will be the fifth revision of the HTML standard originally created in 1990. The key new feature for supporting pub/sub will be the WebSocket interface that allows servers to send asynchronous content to Web browsers. The protocol part of the WebSocket is standardized by the IETF. The Server-sent events is a related specification also part of the HTML5 for providing push notifications from a server to a browser client in the form of DOM events.

PubSubHubbub is an open protocol for distributed pub/sub communication on the Internet. The protocol extends the Atom (and RSS) protocols for data feeds. The main purpose is to provide near-instant notifications of change updates, which would improve on the typical situation where a client periodically polls the feed server at arbitrary interval.

RSS, XMPP, Comet, HTML5, and Pubsubhubbub all introduce pub/sub service features for Web applications. The features are basic in the sense that typically only topic or channel based semantics are supported. The next generation of Web-based pub/sub services is expected to introduce more sophisticated features for supporting Web application development. HTML5 will play a crucial role in supporting the development and deployment of these services.

1.9.4 A Taxonomy

In this section, we present a taxonomy of pub/sub systems that will then be revisited later in the book. We start with a brief survey of taxonomies, and then focus on the taxonomy used in this book.

Typically applications are developed based on modular units of computation, such as classes, modules, and programs. The typical ways of combining the basic units is realized with either explicit or implicit invocation [34, 35]. With explicit invocation, component names are statically bound to the implementations, for example a function invoking another function in some other module via local function call or RPC. This is contrasted by implicit invocation, in which a component publishes an event that then triggers the invocation of the requested functionality. Implicit invocation can abstract the name and location of the component that will perform the requested functionality.

[4] www.twitter.com.
[5] www.facebook.com.

Notkin *et al*. were the first to consider the implicit invocation model in detail. They presented design considerations when introducing implicit invocation to traditional programming languages [35]:

- Event declaration that pertains to the vocabulary that is used to define events and to the properties of the vocabulary.
- Event parameters and attributes, which are about the information associated with an event.
- Event bindings, which determine how events are bound to components that process them.
- Event announcement, which determines the invocation model: explicit or implicit.
- Delivery policy defines the rules for event delivery.
- Concurrency pertains to the number of threads and their priorities.

The design consideration for implicit invocation contain the relevant issues from the viewpoint of programming languages; however, they are not sufficient for the development of distributed systems. The distributed environment has been investigated in the distributed pub/sub and event processing community. An early taxonomy of the area is given in [36] and later taxonomies are given in [1] and [37].

We now summarize the history and evolution of publish/subscribe with a taxonomy that demonstrates differences between the ideas and systems considered so far. Figure 1.12 presents the taxonomy that has three core topics, namely research, standards, and Web. In the research category, we have two subcategories, systems and patterns. The systems subcategory contain concrete research proposals, and the pattern subcategory contains abstract architectural and design patterns. Patterns are used to design and implement pub/sub systems. Patterns are also heavily used in the design of standardized solutions, for example the CORBA Event Service and Notification Service are based on the event channel pattern.

The research systems category has two subcategories: centralized/cluster-based solutions, and wide-area systems. The former includes the LINDA, ISIS, and SIFT systems,

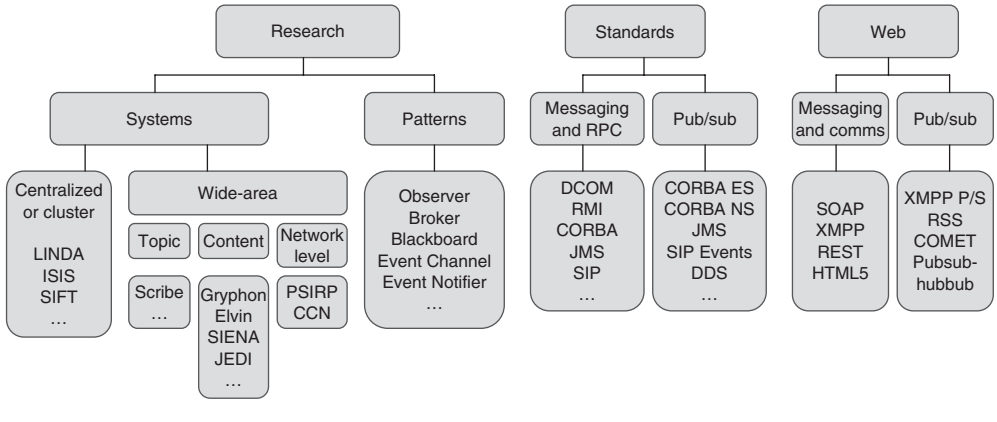

Figure 1.12 A taxonomy of pub/sub solutions.

and the latter include various wide-area systems. The wide-area systems differ based on the operating semantics. We have identified three subcategories, namely topic-based, content-based, and network layer systems. Scribe is an example of a topic-based DHT, SIENA is the classical content-based system, and PSIRP and CCN are examples of recent network layer systems.

The standards category has two subcategories, namely messaging and RPC, and pub/sub. The former contains systems and APIs such as DCOM, RMI, and JMS. The latter contains pub/sub standards, such as CORBA Event Service and Notification Service, JMS, and SIP events.

The Web category has also two subcategories: messaging and communications, and pub/sub. Basic techniques in the first subcategory include SOAP, XMPP, REST, and HTML5. The second subcategory contains XMPP pub/sub, RSS, and other pub/sub related proposals.

The presented taxonomy is coarse grained and can be extended with various functional and nonfunctional details, such as Quality-of-Service support, reliability, fault-tolerance, composite event detection support, state aggregation support, mobility support, etc. We will return to many of these features later in the book.

1.10 Application Areas

In this section, we briefly examine a number of pub/sub application areas. We return to the applications towards the end of the book in Chapter 12 and examine them in more detail. This analysis of the applications will consider how the patterns, protocols, and solutions covered in this book are applied in current applications.

As mentioned, pub/sub solutions can be applied widely in both local and distributed environments. Typical pub/sub applications include the following:

- GUIs, in which pub/sub is typically applied as the glue that connects the various components together. A classical example is the MVC design pattern heavily used in GUIs and its component the observer pattern. These patterns will be presented in more detail later. They allow application components to react to various events, such as a user pressing a touch screen.
- Information push, in which content is pushed to the user. This is a fundamental requirement for many applications that rely on real-time or near real-time data.
- Information filtering and targeted delivery used by alert and presence services (Google Alerts etc.), application stores, RSS brokering services, etc. Examples include XMPP Pub/sub, Pubsubhubbub, Facebook Messenger and Chat, and Twitter.
- Signalling plane, in which pub/sub ensures that asynchronous events are delivered in real-time or near real-time from publishing components to subscribing components. Example applications include industrial and tactical systems. DDS is the key standard for these systems.
- *Service Oriented Architecture (SOA)* and business applications rely on pub/sub in the Enterprise Service Bus (ESB). The ESB is typically implemented with an XML message broker.
- *Complex Event Processing (CEP)* for data analysis. CEP is used extensively in various business applications, for example algorithmic trading and fault detection.

- Cloud computing, in which pub/sub and message queuing is used to connect the different cloud components together.
- Internet of Things, in which pub/sub connects the sensors and actuators together and with Internet resources.
- Online multiplayer games, in which pub/sub is used to synchronize game state across players and servers.

It is evident that pub/sub is employed in differing environments and applications. Thus many different flavours of pub/sub are needed to solve the information dissemination problem for specific environments.

1.11 Structure of the Book

This book examines pub/sub technology and its applications. Our focus is on the design of such systems based on modular building blocks and commonly employed solutions. In order to examine the building blocks and assess their applicability to various scenarios, we first investigate the history of pub/sub systems and how the different solutions have been developed and deployed.

The pub/sub functionality is typically offered in the form a library or middleware service that applications can then utilize. Pub/sub can also be realized on lower layers of the protocol stack as well as in the application itself. Many of the solutions for the the distributed environment are based on overlay networks that are networks that operate on top of the TCP/IP protocol stack. Therefore we address the middleware and overlay systems in detail, but also consider new proposals that introduce pub/sub features in the protocol stack.

Throughout the book the examination includes three differing views to pub/sub, namely standard-based solutions designed to solve specific industrial cases, Web-based solutions developed for the Internet, and research-oriented solutions that consider potential future applications of the technology. These three views are overlapping, and in many cases solutions developed in the academia end up being adopted by specific standards.

Chapter 2 examines the basic technology for supporting distributed pub/sub systems. We consider TCP/IP, naming and addressing, firewalls and NAT devices, and advanced techniques such as multicast, causality, and messaging. TCP/IP as well as store-and-forward messaging solutions are the foundation for pub/sub solutions. We consider two standards based interoperable messaging frameworks, namely Web services and SIP. This chapter provides the essential networking and messaging technology background for later chapters.

Chapter 3 deepens the treatment of networking solutions by illustrating how networks can be created on top of networks in so called overlay solutions. Overlay networks are a robust and scalable solution that does not require the changing of routers or the basic protocol stack. Thus overlay networks are good candidates for supporting various distributed pub/sub systems. We consider Distributed Hash Tables (DHTs) that are a specific overlay solution that have many promising application in information dissemination and content delivery. The DHT-based solutions are examined in subsequent chapters with more details.

Chapter 4 considers the key design principles and patterns for pub/sub systems. We give an overview of the environment, principles, and patterns and then examine pub/sub specific patterns in more detail. Key patterns covered in this chapter are the observer, and

event notifier patterns. Indeed, the event notifier pattern is the basis for the distributed pub/sub systems covered in later chapters.

Chapter 5 presents key pub/sub standards and specifications as well as messaging products. We investigate standards such as CORBA Event Service and Notification Service, OMG DDS, SIP Event Framework, JMS and product technologies such as COM+ and.NET, Websphere MQ, and AMQP. The standards exemplify many of the patterns introduced in the previous chapter.

Chapter 6 examines state of the art Web technologies for building pub/sub systems over the Web. These technologies include REST, AJAX, RSS and Atom, SOAP, and XMPP. Web-based protocols are necessary for the creation of real-time and interactive Web pages and applications, and thus an integral part of modern Web application development.

Chapter 7 considers the distributed pub/sub environment in general and presents a number of solutions for meeting various information dissemination requirements. The chapter examines various routing functions including topic, content, and gossip based mechanisms as well as optimization techniques such as filtering, filter covering, and filter aggregation with merging. The chapter together with the following chapters present a toolkit of solutions that can be applied by developers in order to engineer efficient distributed pub/sub solutions.

Chapter 8 investigates content matching techniques and efficient filtering solutions. Content matching is a basic requirement for a pub/sub system and thus it should be efficient and scalable as well as support different filtering constraints. We investigate well-known techniques including counting based algorithms and the poset and forest algorithms.

Chapter 9 examines well-known research prototypes and solutions for pub/sub. The solutions incorporate many of the patterns and solutions introduced in previous chapters, such as the event notifier pattern, filter matching, and filter covering and merging. We consider classical examples such as SIENA, Gryphon, JEDI, Elvin as well as more recent DHT-based solutions. This chapter provides specific examples of systems that utilize solutions presented in previous chapters.

Chapter 10 presents a concrete example of a keyword-based pub/sub system implemented on top of a DHT overlay network. The chapter considers the complexity of the problem and presents an efficient solutions based on rendezvous points on the DHT based network. The chapter illustrates how the already discussed DHT-based solutions can be utilized for keyword based content dissemination.

Chapter 11 consider advanced features of pub/sub systems. We start a number of security solutions for pub/sub. Then we examine topics such as composite subscriptions, filter merging, load balancing, channelization, reconfiguration, mobility support, congestion control, and the evaluation of pub/sub system. Many of the topics pertain to the pub/sub routing topology, its organization and configuration.

Chapter 12 considers applications of pub/sub systems and technology. We consider the role of pub/sub as an enabler of a cloud computing platform, a generic XML-broker for enterprise applications, content advertisement with pub/sub technologies, SOA, CEP, and several Web based applications including Pubsubhubbub, Facebook, and the Apple push service for mobile devices. The patterns and solutions used by the applications are discussed.

Chapter 13 considers new research proposals in adopting the pub/sub paradigm in proposed new protocol architectures that replace TCP/IP with receiver driven protocol suites. The motivation, features, and possibilities of these systems are discussed.

Chapter 14 presents a summary and conclusions of the book. We discuss the role of pub/sub technology as a generic enabler for connecting components across space, time, and synchronization in vast distributed systems.

References

1. Baldoni R, Querzoni L, Tarkoma S and Virgillito A (2009) Distributed event routing in publish/subscribe communication systems. *MiNEMA State-of-the-Art Book*. Springer.
2. Hinze A and Buchmann AP (eds) (2010) *Principles and Applications of Distributed Event-Based Systems*. IGI Global.
3. Tarkoma S (2010) *Overlay Networks – Toward Information Networking*. CRC Press.
4. Eugster PT, Felber PA, Guerraoui R and Kermarrec AM (2003) The many faces of publish/subscribe. *ACM Comput Surv* **35**(2), 114–31.
5. Carriero N and Gelernter D (1989) Linda in context. *Commun ACM* **32**(4), 444–58.
6. Sun (2002) *Java Message Service Specification 1.1*.
7. Pietzuch P, Eyers D, Kounev S and Shand B 2007 Towards a common api for publish/subscribe *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pp. 152–157 DEBS '07. ACM, New York, NY, USA.
8. Carzaniga A, Rosenblum DS and Wolf AL (2001) Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* **19**(3), 332–83.
9. Colouris G, Dollimore J and Kindberg T (1994) *Distributed Systems: Concepts and Design*, 2nd edn. Addison-Wesley, Boston, Massachusetts.
10. Object Computing, Inc. (2001) *CORBA Notification Service Specification v.1.0*. OCI.
11. Fleisch BD (1987) Distributed shared memory in a loosely coupled distributed system. *SIGCOMM Comput Commun Rev* **17**, 317–27.
12. Hayes-Roth B (1985) A blackboard architecture for control. *Artificial Intelligence* **26**(3), 251–321.
13. Krasner GE and Pope ST (1988) A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program*. **1**, 26–49.
14. Birman KP and Joseph TA (1987) Reliable communication in the presence of failures. *ACM Transactions on Computer Systems* **5**, 47–76.
15. Oki BM, Pflügl M, Siegel A and Skeen D (1993) The information bus – an architecture for extensible distributed systems. Proceedings of the Fourteenth ACM Symposium on Operating System Principles, 5–8 December 1993, Asheville, North Carolina, pp. 58–68.
16. Yan TW and Garcia-Molina H (1999) The SIFT information dissemination system. *ACM Transactions on Computer Systems Database Systems* **24**, 529–65.
17. IBM (2002) *Gryphon: Publish/subscribe over public networks*. (White paper) http://researchweb.watson.ibm.com/distributedmessaging/gryphon.html.
18. Strom RE, Banavar G, Chandra TD, et al. (1998) Gryphon: An information flow based approach to message brokering. *Computing Research Repository (CoRR)*. Available at: http://arxiv.org/corr/home.
19. Sutton P, Arkins R and Segall B (2001) Supporting disconnectedness-transparent information delivery for mobile and invisible computing *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, p. 277. IEEE Computer Society, Washington, DC, USA. 19
20. Castro M, Druschel P, Kermarrec AM and Rowstron A (2002) Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, **20**(8): 1489–99.
21. Mühl G, Ulbrich A, Herrmann K and Weis T (2004) Disseminating information to mobile clients using publish-subscribe. *IEEE Internet Computing* **8**, 46–53.
22. Pietzuch PR (2004) *Hermes: A Scalable Event-Based Middleware*. PhD thesis. Computer Laboratory, Queens' College, University of Cambridge.

23. Pietzuch PR and Bacon J (2002) Hermes: A distributed event-based middleware architecture *ICDCS Workshops*, pp. 611–18.
24. Carzaniga A, Rutherford MJ and Wolf AL (2004) A routing scheme for content-based networking. *Proceedings of IEEE INFOCOM 2004*. IEEE, Hong Kong, China, vol. 2, pp. 918–28.
25. Cugola G, Di Nitto E and Fuggetta A (1998) Exploiting an event-based infrastructure to develop complex distributed systems *Proceedings of the 20th International Conference on Software Engineering*, pp. 261–70. IEEE Computer Society.
26. Gritter M and Cheriton DR (2001) An architecture for content routing support in the Internet. *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems – Volume 3*, p. 4, USITS'01. USENIX Association, Berkeley, CA.
27. Koponen T, Chawla M, Chun BG (2007) A data oriented (and beyond) network architecture. *SIGCOMM Comput. Commun. Rev*. **37**(4), 181–92.
28. Tarkoma S, Ain M and Visala K (2009) The Publish/Subscribe Internet Routing Paradigm (PSIRP): designing the future Internet architecture. *Future Internet Assembly*, pp. 102–111.
29. Jacobson V, Smetters DK, Thornton JD, Plass MF, Briggs NH and Braynard RL (2009) Networking named content. *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, pp. 1–12. CoNEXT '09. ACM, New York, NY, USA.
30. Object Computing, Inc. (2001) *CORBA Event Service Specification v.1.1*. OCI.
31. Rosenberg J, Schulzrinne H, Camarillo G, et al. (2002) *RFC 3261: SIP: Session Initiation Protocol*. IETF. http://www.ietf.org/rfc/rfc3261.txt.
32. Object Computing, Inc. (2007) *Data Distribution Services, V1.2*. OCI.
33. Saint-André P (2004) *RFC 3920: Extensible Messaging and Presence Protocol (XMPP): Core*. Internet Engineering Task Force.
34. Garlan D, Jha S, Notkin D and Dingel J (1998) Reasoning about implicit invocation. *SIGSOFT Softw. Eng. Notes* **23**, 209–21.
35. Notkin D, Garlan D, Griswold WG and Sullivan KJ (1993) Adding implicit invocation to languages: Three approaches. *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pp. 489–510. Springer-Verlag, London.
36. Meier R and Cahill V (2002) Taxonomy of distributed event-based programming systems *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pp. 585–8. ICDCSW '02. IEEE Computer Society, Washington, DC.
37. Blanco R and Alencar P (2010) Event models in distributed event based systems. *Principles and Applications of Distributed Event-Based Systems*, pp. 19–42.