Chapter 6

Coordination and Events

Many applications involve a number of activities that cooperate to achieve a common goal and need to interact in order to coordinate their evolution. In this chapter, we are interested in *asynchronous* interaction based on three related mechanisms: events, i.e. state transitions that trigger reactions, message queuing, and shared data-spaces. We present the main paradigms of asynchronous coordination of loosely coupled activities and their implementation in middleware. These notions are illustrated by a few case studies.

6.1 Introducing Coordination

6.1.1 Motivation and Requirements

In a broad sense, *coordination* refers to the methods and tools that allow several entities to cooperate towards a common goal. A coordination model provides a framework to organize this cooperation, by defining three elements [Ciancarini 1996]: a) the *coordination entities* whose cooperation is being organized, e.g. processes, threads, various forms of "agents" (see 6.6), organizations, people, etc.; b) the *coordination media* through which the entities communicate, e.g. messages, shared variables, or more elaborate mechanisms built above the basic communication layers; and c) the *coordination rules*, which define the interaction primitives and patterns used by the cooperating entities to achieve coordination.

Besides the notion of cooperation, coordination also has a function of integrating previously independent activities. In the words of [Gelernter and Carriero 1992]: "A coordination model is the glue that binds separate activities into an ensemble". In that sense, coordination is related to composition. The main difference is that coordination models are associated with loose coupling, while most component-based models (Chapter 7) imply a notion of strong coupling.

We are specifically interested in application domains that are subject to the following requirements.

- *Loose coupling.* The entities involved in the cooperation should be allowed to evolve independently, and coordination should therefore not impose strong constraints on their behavior.

- *Dynamic evolution.* The cooperating entities are allowed to join and leave the application freely, during execution. Late binding is therefore mandatory.
- *Large scale.* The number of cooperating entities may potentially become very large and they may operate over a wide geographical range; the coordination algorithms and structures should therefore be scalable with respect to these two aspects.
- *Heterogeneity.* The cooperating entities may use heterogeneous hardware, operating systems, run time environments, and security policies; they may be administered by various authorities.

Examples of such application domains include monitoring of industrial installations, supervision of networking equipment, weather forecast using distributed sensors, stock tracking, distributed auctions. In these applications, systems must react to variations in the environment, and notifications of changes must be propagated to a dynamically changing set of recipients, which may themselves react by generating other notifications. Devices and services may be dynamically added or removed. These environments usually do not have a central controlling authority. Another relevant area is that of ubiquitous and mobile systems, which involve spontaneous, usually unpredictable, patterns of communication between heterogeneous objects such as mobile phones, portable or wearable devices, sensors and actuators embedded in mobile equipment, etc. An asynchronous, loosely coupled communication model is well adapted to these requirements.

6.1.2 Terminology

Before introducing the plan of the chapter, a word on terminology is in order, because the terms that designate the main concepts of coordination are heavily overloaded and therefore subject to misuse.

An *event* is a detectable state transition that locally occurs at a definite point in time in a specified environment. Example of events, in the environment of a process, are: the change of the value of a variable (the contents of a memory location); the occurrence of an interrupt, e.g. a timeout expiration signal or a mouse button click.

In a more general sense, in the context of a distributed system, an event is the notification of a local state transition (an event in the above sense) to a set of (usually remote) receiving entities. The entity in which the state transition occurred is an event producer, or *event source*; the entities that receive the notification are event consumers, or *event sinks*. An event notification may be initiated by the source (*push* mode) or by the sink (*pull* mode); many coordination schemes use a combination of these two modes.

Note that the reception of a notification by a consumer is a (local) event, in the original sense, for that consumer; and that event may in turn be notified to other consumers. Also note that the term "notification" designates both the action of signaling an event to consumers and the data structure used for this signaling.

The reception of an event notification by an event sink causes the execution of a specified *reaction*. The form of this reaction, and the association between an event and a reaction, depend on the specific coordination model and on the framework that implements it. The terms *callback* and *handler* are often used in relation to the implementation of reactions to events.

A *message* (as defined in Chapter 4) is an information transmitted by a sender to one or several receiver(s). A message may be considered at different levels of abstraction, from the physical signal to more elaborate formats. In the context of coordination systems, a message usually has a composite structure that includes a contents proper and meta-information whose format depends on the specific messaging system.

From the above definition, events (more precisely event notifications, which propagate the occurrence of a local event) are specific instances of messages. The difference between events and messages is one of usage rather than an intrinsic one; and a number of basic patterns (e.g. the selection of receivers) are common to both mechanisms. One usually refers to "events" when emphasis is on the occurrence and signaling of the initial transition (the local event), and to "messages" when emphasis is on the information contents being transmitted. However, in many cases, the two terms are not clearly distinguished. Products that provides coordination primitives using either form of communication are collectively called *Message-Oriented Middleware* (MOM).

In the context of coordination systems, the term *agent* refers to an entity that coordinates its activity with that of other similar entities, and which has the following characteristics: it is active, i.e. it encapsulates at least one thread of execution; it is self-contained, i.e. it includes a minimal set of resources needed for its execution; it may communicate both with its environment (i.e. the run time system that supports its execution) and with other agents; its behavior is determined both by its own initial program and by its interactions; it may be fixed or mobile (in the latter case, it may migrate from one environment to another one). The term "agent" is mainly used in conjunction with a specific support system, which defines its interaction primitives and patterns.

In the rest of this chapter, we present the principles of these coordination mechanisms and the middleware that implements them. The common patterns for coordination are reviewed in Section 6.2. Event-based communication is the subject of Section 6.3. Message-based communication and middleware are presented in Section 6.4. Coordination through shared objects is discussed in Section 6.5. Some instances of agents are described in Section 6.6. Finally, Section 6.8 is a case study of a message-oriented middleware.

6.2 Patterns for Coordination

The three patterns that follow are directly relevant to coordination. The main common objective is to favor uncoupling between the cooperating entities, which leads to various forms of indirect, asynchronous interaction. Since these patterns apply to several coordination models, and since they may be implemented using various techniques, we present them in an abstract form, concentrating on the communication aspects. Some indications on the implementation techniques and on the non-functional properties are given in the case studies.

6.2.1 Observer

Observer is the basic pattern that underlies coordination. In its primitive form [Gamma et al. 1994], it involves one "observed" object and an unspecified number of independent "observer" objects. The problem is for the observers to be kept aware of any change occurring in the observed object.

This situation typically occurs when a given abstract entity (e.g. the contents of a spreadsheet) has several different representations, or images (e.g. a table, a histogram, a pie chart, etc.). Whenever that entity is modified, all its images need to be updated to reflect these changes; these images, and the processes that maintain them, are independent of each other, and new forms of images may be introduced at any time.

In the proposed solution, the observers register their interest with the observed, which therefore ke a list of these observers. When a change occurs, the observed asynchronously notifies all the registered observers. Each observer can then query the observed to get aware of the changes. This pattern therefore uses both the push and the pull modes.

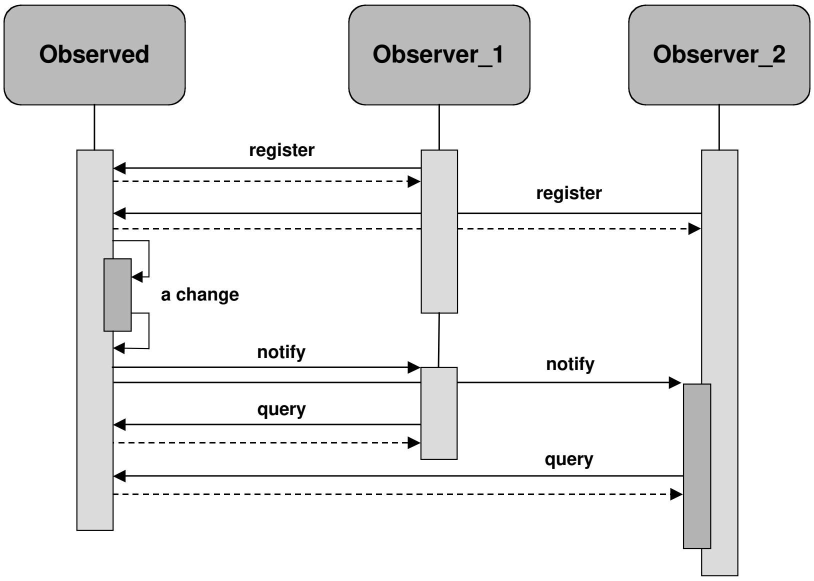

Figure 6.1. The OBSERVER pattern

This solution favors uncoupling, not only between the observers, which are not aware of each other, but also between the observed and the observers. Once an observer has been informed of a change, it may query the observed for a specific piece of information and it may choose the moment of the query.

This pattern has the following limitations.

- The observed entity bears a heavy load, since it has to provide and to implement the registration interface, to keep track of the observers, and to answer the queries. This is detrimental to performance and to scalability.

- While favoring uncoupling, the two-stage process of notification and query is not selective and may lead to unnecessary exchanges (e.g. an observer may not be interested in a specific change, but has to query the observed in all cases).

The following pattern attempts to remedy these drawbacks.

6.2.2 Publish-Subscribe

The PUBLISH-SUBSCRIBE pattern is a more general form of OBSERVER. It defines two "roles", Publisher (event source) and Subscriber (event sink). A given entity may take up either role, or both.

Events generated by publishers are selectively notified to subscribers. The selection is achieved by defining event classes, i.e. sets of events that satisfy some conditions, to be discussed later. A subscriber may register its interest for a class of events by *subscribing* to that class. When a publisher generates an event E, all subscribers that previously subscribed to an event class to which E belongs receive an asynchronous notification of the occurrence of E. Reacting to this notification is locally arranged by each subscriber (e.g. by associating a specific handler with each class of events, etc.).

We present the pattern in an abstract form by defining a *mediator* as an entity that is responsible for registering subscriptions, receiving events, filtering events according to class definitions, and routing them to the interested subscribers (Figure 6.2). Depending on the specific system, the mediator may be implemented in various ways, e.g. by a centralized server, by distributed cooperating servers, or possibly collectively by the publishers (like in OBSERVER). These organizations are discussed and illustrated by examples in Section 6.3.

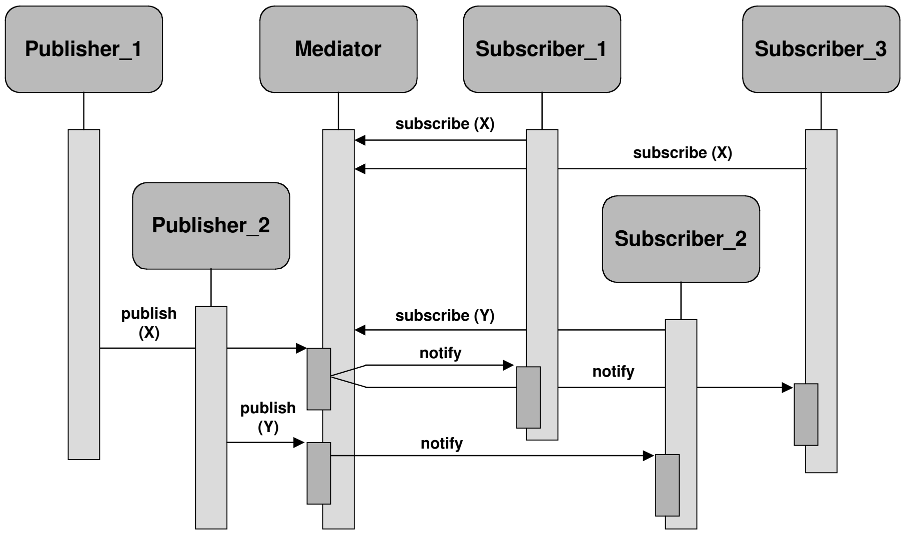

Figure 6.2. The PUBLISH-SUBSCRIBE pattern

There are two main ways of defining event classes.

1. *Topic-based classification.* This scheme relies on the association of events with named topics, or subjects. Topics essentially define a name space (3.1), which may be organized according to the needs of a specific application or application domain. The topics need not be disjoint, i.e. an event may be associated with several topics. Like for any name space, a number of topic organizations may be defined, e.g. flat space, hierarchy, graph, etc., and all the facilities associated with name manipulation

(pattern matching using wildcards, etc.) may be used. The structure of the topic space may conveniently mimic that of other currently used name spaces, e.g. file systems or URLs.

A variation of the topic-based classification is *type-based classification* [Eugster et al. 2000]. In this scheme, events are filtered according to their type, which is a specific attribute. This implies that the publish-subscribe system should be integrated in a typed programming language. Event notifications are then defined as typed objects, with the benefits of typing (safety, compile time checking). Note however that some languages allow dynamic type changes, which offsets these benefits to favor flexibility.

2. *Content-based classification.* This classification scheme introduces an additional degree of flexibility, since events are classified according to their run-time properties, and not to a statically defined topic organization. The "contents" of an event is the data structure used for notification, or meta-data associated with this structure. Event filtering is now an associative process, i.e. matching a specified template against the contents. Depending on the organization of an event's contents, the template may take various forms, e.g. a string, one or several name-value pair(s), etc.

Content-based classification is more general than topic-based classification, since the latter may be implemented by means of the former while the reverse is not true. However, implementing a scalable content-based filtering is a difficult task (see example in 6.3).

The PUBLISH-SUBSCRIBE pattern has the potential of achieving the uncoupling of the cooperating activities:

- due to the indirect interaction scheme, the activities have no knowledge of each other; they do not directly interact for synchronization, and do not need to be active at the same time; they may be dynamically created and removed, and may freely enter or leave the application.
- due to the event filtering mechanism, the number of notifications is reduced: an activity is only notified of the events that it considers relevant; content-based filtering allows this selection to be dynamic.

However, actually achieving these potential benefits is strongly dependent on implementation. For instance a centralized implementation may reduce the mutual independence of activities; or an inefficient implementation of a filtering algorithm may offset the advantage of selectivity. Implementation issues are examined in Section 6.3.

A survey of various aspects of PUBLISH-SUBSCRIBE is given in [Eugster et al. 2003].

6.2.3 Shared Dataspace

The SHARED DATASPACE pattern has been introduced in the Linda language [Carriero and Gelernter 1989, Gelernter and Carriero 1992], under the name of *generative coordination.* It defines a form of communication between uncoupled processes, through a common, persistent information space organized as a set of *tuples*. In the original model, a tuple is a record, i.e. an ordered set of typed fields. A process may insert a new tuple into the space, lookup the space for a tuple matching a specified template (e.g. by specifying the values of some fields) and then read the tuple and possibly remove it from the space. Tuples are persistent, i.e. a tuple remains in the space until explicitly removed. This is a pure pull model in the sense that changes in the tuple space are not explicitly notified to processes. A process whose query fails remains blocked until a tuple matching the query is inserted into the the space.

In the original model, processes are themselves elements of the tuple space. When a process is created, it is inserted in the tuple space; after the process finishes execution, it turns into a passive tuple, which is a representation of its state at the end of the execution.

This pattern has several limitations, which we present together with some proposed remedies.

- There is a single dataspace. Multiple dataspaces are useful for improving modularity, for exploiting locality, and for implementing isolation and protection. Various forms of multiple dataspace organizations (independent, nested) have actually been introduced in several implementations.

- When several tuples match its template, a query returns one of these tuples, and the choice is nondeterministic. When several processes query the space with the same template, there is no way for either of them to find all the tuples that match the template.

  This problem is solved by introducing an additional primitive, *copy-collect*, which allows a process to get *all* the tuples that match a given template.

- The original model is based on a pure pull mode and therefore lacks reactivity. Several implementations have added a *notify* primitive, which signals the introduction of a tuple matching a template to interested processes.

The shared dataspace pattern is useful in applications in which the resolution of a problem is partitioned among a number of "worker" processes, which use the tuple space to allocate work and to share results. Another application area is resource discovery (3.2.3), a form of trading in which resources are looked up according to various criteria, and may be dynamically added and removed.

6.2.4 Extra-Functional Properties

Recall (2.1.3) that extra-functional properties are those which do not explicitly appear in the interface of a service. Several such properties, listed below, are useful additions to a coordination system.

- Persistence. An event or a message may be transient, i.e. it is lost if no receiver is ready to accept it, or it may be persistent, i.e. it is conserved by the system until explicitly removed. Persistence is implemented in most coordination systems because it favors uncoupling, since there is no synchronization constraint between a sender and a receiver.

- Ordering. Message ordering is discussed in **??**. Imposing strong ordering constraints goes against uncoupling. However, such guarantees as causal ordering are unavoidable whenever messages and events are used to maintain such constraints as consistency of replicated information. Designing scalable ordering algorithms is a challenging task because it involves global synchronization, and because large scale communication systems such as the Internet do not provide any ordering guarantees. One approach consists in partitioning the set of coordinating entities and trying to derive global ordering properties from local ones by an adequate choice of the partition (see e.g. [Laumay et al. 2001]).
- Transactions. Executing reactions to events as transactions may be required to preserve application-specific invariants, or to prevent an observer from seeing an inconsistent state of the system.

6.2.5 Comparing the Coordination Patterns

The Publish-Subscribe and Shared Dataspace patterns illustrate two dual approaches to coordination of loosely coupled entities. The first one is based on messages, the second one on shared information. It is interesting to compare their expressive power. The main result in this area has been proved in [Busi and Zavattaro 2001], and may be summarized as follows.

- The Shared Dataspace model may be reduced to (i.e. simulated by) the Publish-Subscribe model.
- The Publish-Subscribe model may not be reduced to the original Shared Dataspace model. However, the reduction becomes possible if the dataspace model is extended with a primitive such as *copy-collect* (6.2.3).

Therefore the choice between the two forms of coordination is not mainly based on expressive power, but on the adequation of the pattern to the main orientation of the problem (event-driven vs. data-driven), and on the availability of efficient and scalable implementations.

6.3 Event-based Communication

In this section, we examine the main implementation aspects of event-based communication based on the Publish-Subscribe pattern. Efficient implementation is critical for achieving the benefits of uncoupling. The most important issue is *scalability*, with respect to both the number of cooperating entities and geographical size.

We first examine three main aspects: organizing the mediator, implementing filtering, implementing notification routing. Since the mediator implements the filtering and routing algorithms, the issue of mediator organization is closely related to the design of these algorithms. For presentation, we find it more convenient to discuss architectural aspects first. In addition, some properties, such as fault tolerance and locality, directly depend on the architecture and can be examined in that context.

This section concludes with a brief review of a few case studies.

6.3.1 Event Mediator Architecture

We assume that the event system actually uses a mediator. In other words, we do not consider systems using the OBSERVER pattern, because the requirement of uncoupling precludes direct access from the consumers to the producers[1].

The mediator is the entity that receives events from publishers and delivers notification to subscribers. Implementing the mediator as a single server has the usual drawbacks of centralized architectures: the server is a single point of failure and a performance bottleneck. This solution is limited to small scale system on local area networks.

Event mediators for large scale systems are therefore organized as multiple cooperating servers. There are several ways of designing the server architecture.

1. The first solution is a hierarchical organization, in which the servers are organized into a tree. Coordinating entities (publishers and subscribers) may be connected to any server in the hierarchy. If a server does not have the information to process a request (the distribution of information among the servers depends on the algorithms), it sends it to its parent server.

   While the communication pattern is straightforward, this solution has two main drawbacks: it puts a heavy load on the higher levels of the hierarchy, and it is sensitive to server failures, since a failed server isolates all its descendants. The usual remedy for these drawbacks, server replication, is not easily applicable because the information update rate is typically high, and therefore replica consistency is expensive to maintain.

2. The second solution is an acyclic peer to peer organization, in which the communication pattern between servers is an acyclic undirected graph. The main benefit over the hierarchical organization is a better distribution of the load. The absence of cycles in the communication pattern can again be exploited by the filtering and notification algorithms (e.g. algorithms based on a spanning tree are easily implemented).

3. The third, more general solution is a general peer to peer organization, using an unrestricted graph communication pattern. This solution has no limitations and may be used to improve fault tolerance by redundancy. However, the communication graph has no special properties.

These organizations are discussed in more detail in [Carzaniga et al. 2001].

As usual, hybrid solutions combining the advantages of the above "pure" communication schemes are found in practice. An interesting solution is a clustered servers organization in which inter-cluster communication follows an acyclic pattern, while intra-cluster communication is unrestricted. This organization ke the advantage of acyclicity for inter-cluster (global) communication and has the following additional benefits.

- Exploiting locality. In many applications, the coordinated entities are grouped according to organizational or geographic criteria, and the communications within a

[1] An example of an event model using direct access is the Java Distributed Event model used in JavaBeans, a system essentially used in a centralized environment.

group are more frequent than those between groups. Server clusters can be organized so as to reflect this structure, making intra-group communication more efficient.

- Improving fault tolerance. Server availability is increased by using some nodes of a cluster as replicas. Replica consistency must only be maintained within a cluster.
- Allowing efficient causal delivery. Causal delivery of notifications is discussed in 6.2.4. As shown in [Laumay et al. 2001], partitioning the servers in acyclically connected clusters allows a scalable implementation of causal delivery.

6.3.2 Event Filtering

Event filtering (or matching) is the process of finding the subscriptions that match a published event, so as to select the recipients of notification. The problem is different for the two forms of subscription described in 6.2.2.

- For topic-based subscription, the topics are statically known, before the event is published. The problem reduces to looking up an element in a (possibly distributed) table. A structured organization of the topics (e.g. hierarchical) allows for faster search. In any case, scalable lookup algorithms based on hash-coding are known.
- For content-based subscription, the situation is different, because the contents associated with an event is only known at publication time. Designing scalable algorithms for this case is still a research issue.

Here we only consider content-based subscription. A naive algorithm would match the contents of a publish message against all registered subscriptions, and would therefore run in linear time with respect to the number of subscriptions. For Internet-scale systems, this is considered inefficient, and the goal is to design sub-linear matching algorithms.

Such an algorithm, used in the Gryphon system [Gryphon 2003], is described in [Aguilera et al. 1999]. It is based on pre-processing of subscriptions. The motivation is that the rate of change of subscriptions is slow with respect to the rate of event publishing, which makes pre-processing feasible. Pre-processing creates a matching tree, in which each node describes a test on some subscription attribute, and the edges outgoing from this node represents the results of this test. The leaves of the tree represent individual subscriptions. Thus an event is matched against successive nodes of the tree, starting from the root, by performing the prescribed tests. Several paths are possible, i.e. an event may match several subscriptions.

The matching algorithm is used on each server, in a multi-server mediator system. The problem of routing events in the network of servers is examined in the next section.

6.3.3 Notification Diffusion

In the trivial case in which there is no filtering (or a subscription that matches all events), the problem of routing reduces to multicasting a notification to all subscribers. This problem is adequately solved, even on a large scale, by protocols such as IP Multicast (see Chapter 4).

The solution is again easy for topic-based subscription. A multicast group is associated with each different topic; new subscribers join the relevant groups, and events published under a topic are notified to the associated group, again using a group multicast protocol.

With content-based subscription, in a publish-subscribe system implemented as a network of servers (6.3.1), the question is how to best exploit the topology of the network. There are two extreme solutions, assuming that each publisher or subscriber sends its request to one server in the network.

- Propagating each subscription to all servers, and performing the matching at the server on which the event is published. This produces a list of matching subscribers, to which a notification is multicast.

- Registering each subscription on the server that receives it, and propagating all published events to all servers. Matching is then performed on the servers on which subscriptions have been received.

The first solution is acceptable in small scale systems (recall that the rate of subscriptions is usually much lower that that of event production). Neither solution scales well, however. Therefore more elaborate solutions have been proposed.

In Gryphon, a server which receives an event uses the matching tree pre-computed from subscriptions (6.3.2), in order to selectively propagate notifications to its neighbors (on a spanning tree covering the network of servers). The notifications are only sent on those links that are part of a path to at least one matching subscriber. The algorithm is described in [Banavar et al. 1999].

Siena uses routing algorithms based on two general requirements, inspired by the design principles of IP Multicast: a) to route notifications in one copy as far as possible (i.e. to start replicating it only as close as possible to its destinations); and b) to apply filtering as close as possible to the event sources. Thus a subscription sets up a routing path to be used later to propagate notifications, in the form of a tree that connects the subscriber to all servers in the network. Notifications that match the subscription are then routed to the subscriber following the routing path in the reverse direction. This may be again optimized if event sources give advance information, in the form of *advertisements*, on the classes of events that they will publish.

6.3.4 Examples

Currently not available.
The examples should include:
Gryphon [Banavar et al. 1999] [Aguilera et al. 1999]
Siena [Carzaniga et al. 2001] [Rosenblum and Wolf 1997]
Infobus [Oki et al. 1993] JEDI [Cugola et al. 2001]

6.4 Message Queuing

6.4.1 Message Queuing Principles

We now consider a coordination mechanism based on message queuing. With respect to the general paradigm of message passing as presented in Chapter 4, the specific feature of this mechanism is *indirect* communication: messages are not directly sent to receivers, but to named queues, from which the receivers can retrieve them. As a consequence:

- There is no need for synchronization between senders and receivers. The receiver of a message need not be active when the message is sent, and the sender need not be active when the message is received.
- Messages are persistent; a message is conserved by the system until explicitly removed from a queue.

These properties favor uncoupling between senders and receivers.

A message queue is a named, persistent container, in which messages are placed and conserved until they are removed. Message queuing achieves a function similar to that of topic-based subscription, i.e. organizing messages according to specified centers of interest.

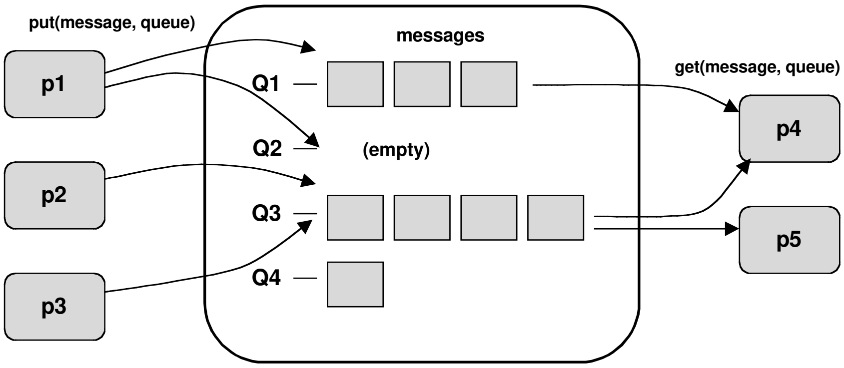

Figure 6.3. Message queuing

A message queuing system provides primitives to put messages in queues and to retrieve messages from queues (Figure 6.3). The latter operation is provided in many variants: it may be blocking or non blocking, and it may remove the message from the queue or leave it there (e.g. to be read by another receiver).

In its pure form, a message queuing system is pull only; it may be augmented with notifications, i.e. a receiver may subscribe to a queue name, and be notified when a new message is put into the designated queue.

6.4.2 Message Queuing Implementation

The main design principle of message queuing implementation is location transparency: senders and receivers see the message queues as logical entities, and the mapping of the queues to physical locations is invisible. This is consistent with the above mentioned analogy between message queues and topic-based publish-subscribe.

A message queuing system may be implemented by a central server that manages the queues, or (more usually) by a network of message brokers.

6.4.3 Examples

Currently not available.

6.5 Coordination through Shared Objects

Currently not available.

6.6 Active Objects and Agents

Currently not available.

6.7 Web Services Coordination

Currently not available.

6.8 Case Study: Joram, a Message-Oriented Middleware

Currently not available.

6.9 Historical Note

Asynchronous events are the earliest coordination mechanism, since the first systems were developed at a low level, directly using the hardware interface, and relied on the hardware interrupt mechanism. In the absence of high-level structures and primitives, these systems were fragile and error prone; due to the transient nature of interrupts, many errors were difficult to reproduce and therefore hard to correct.

As a consequence, the first attempts towards higher level coordination mechanisms aimed at masking the transient and asynchronous nature of interrupts, through such devices as "wake-up waiting switch" and other forms of interrupt memorization. In the mid-1960s, the concepts of sequential process and synchronization were established and led to the invention of semaphores and monitors.

Apart from low-level interrupt handling, which was still used in real-time settings, message passing remained the main asynchronous coordination device in the 1970s. Messages and processes were recognized to have a similar expressive power [Lauer and Needham 1979]. However, most efforts were directed towards the development of tools based on synchronous communication. Attempts towards developing asynchronous communication models, (e.g. [Hewitt 1977] in the context of artificial intelligence), did not meet wide application needs at that time.

In the 1980s, the emergence of new application domains favored the use of asynchronous models based on an event-reaction scheme: graphical user interfaces (GUI), tool integration in software development environments [Reiss 1990], triggers in databases. This period was also marked by a better understanding of the notions of event ordering, causal dependence, and fault tolerance. In the mid 1980's, research on distributed objects introduced the notion of a shared distributed information space. Coordination based on shared dataspaces [Gelernter 1985] was also developed at that time, in a different context.

In the early 1990s, the fast development of local area networks (LANs) stimulated the development of coordination tools based on these networks, such as publish-subscribe systems [Oki et al. 1993] and message queues. In the late 1990s, new application needs such as system administration and networking equipment supervision increased the need for scalable asynchronous coordination. The current challenge is to develop efficient and reliable event distribution systems that can be deployed on the global Internet and that can be augmented with additional properties such as event ordering and transactional processing.

References

[Aguilera et al. 1999] 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 *Symposium on Principles of Distributed Computing (PODC)*, pages 53–61.

[Banavar et al. 1999] Banavar, G., Chandra, T. D., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. (1999). An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 262–272, Austin, Texas.

[Busi and Zavattaro 2001] Busi, N. and Zavattaro, G. (2001). Publish/subscribe vs. shared dataspace coordination infrastructures. In *In Proc. of IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'01)*, pages 328–333. IEEE Press.

[Carriero and Gelernter 1989] Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

[Carzaniga et al. 2001] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383.

[Ciancarini 1996] Ciancarini, P. (1996). Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302.

[Cugola et al. 2001] Cugola, G., Nitto, E. D., and Fuggetta, A. (2001). The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850.

[Eugster et al. 2003] Eugster, P. Th., Felber, P., Guerraoui, R., and Kermarrec, A.-M. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131.

[Eugster et al. 2000] Eugster, P. Th., Guerraoui, R., and Sventek, J. (2000). Type-Based Publish/Subscribe. Tech. Report DSC ID 2000-029, École Polytechnique Fédérale de Lausanne, Laboratoire de Programmation Distribuée.

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 416 pp.

[Gelernter 1985] Gelernter, D. (1985). Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112.

[Gelernter and Carriero 1992] Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107.

[Gryphon 2003] Gryphon (2003). The Gryphon System. http://www.research.ibm.com/gryphon/.

[Hewitt 1977] Hewitt, C. (1977). Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8(3):323–364.

[Lauer and Needham 1979] Lauer, H. C. and Needham, R. M. (1979). On the Duality of Operating Systems Structures. In *Proc. Second International Symposium on Operating Systems,* IRIA, Oct. 1978, volume 13 of *ACM Operating Systems Review*, pages 3–19.

[Laumay et al. 2001] Laumay, Ph., Bruneton, É., De Palma, N., and Krakowiak, S. (2001). Preserving Causality in a Scalable Message-Oriented Middleware. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg.

[Oki et al. 1993] Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. (1993). The Information Bus – An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, volume 27 of *Operating Systems Review*, pages 58–68, Asheville, NC (USA).

[Reiss 1990] Reiss, S. P. (1990). Connecting Tools Using Message Passing in the Field Program Development Environment. *IEEE Software*, 7(4):57–66.

[Rosenblum and Wolf 1997] Rosenblum, D. S. and Wolf, A. L. (1997). A Design Framework for Internet-Scale Event Observation and Notification. In Jazayeri, M. and Schauer, H., editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 344–360. Springer–Verlag.