Chapter 13. Distributed Coordination-Based Systems

In the previous chapters we took a look at different approaches to distributed systems, in each chapter focusing on a single data type as the basis for distribution. The data type, being either an object, file, or (Web) document, has its origins in nondistributed systems. It is adapted for distributed systems in such a way that many issues about distribution can be made transparent to users and developers.

In this chapter we consider a generation of distributed systems that assume that the various components of a system are inherently distributed and that the real problem in developing such systems lies in coordinating the activities of different components. In other words, instead of concentrating on the transparent distribution of components, emphasis lies on the coordination of activities between those components.

We will see that some aspects of coordination have already been touched upon in the previous chapters, especially when considering event-based systems. As it turns out, many conventional distributed systems are gradually incorporating mechanisms that play a key role in coordination-based systems.

Before taking a look at practical examples of systems, we give a brief introduction to the notion of coordination in distributed systems.

13.1. Introduction to Coordination Models

Key to the approach followed in coordination-based systems is the clean separation between computation and coordination. If we view a distributed system as a collection of (possibly multithreaded) processes, then the computing part of a distributed system is formed by the processes, each concerned with a specific computational activity, which in principle, is carried out independently from the activities of other processes.
[Page 590]

In this model, the coordination part of a distributed system handles the communication and cooperation between processes. It forms the glue that binds the activities performed by processes into a whole (Gelernter and Carriero, 1992). In distributed coordination-based systems, the focus is on how coordination between the processes takes place.

Cabri et al. (2000) provide a taxonomy of coordination models for mobile agents that can be applied equally to many other types of distributed systems. Adapting their terminology to distributed systems in general, we make a distinction

between models along two different dimensions, temporal and referential, as shown in Fig. 13-1.

Figure 13-1. A taxonomy of coordination models (adapted from Cabri et al., 2000).

Referential \ Temporal	Coupled	Decoupled
Coupled	Direct	Mailbox
Decoupled	Meeting oriented	Generative communication

When processes are temporally and referentially coupled, coordination takes place in a direct way, referred to as direct coordination. The referential coupling generally appears in the form of explicit referencing in communication. For example, a process can communicate only if it knows the name or identifier of the other processes it wants to exchange information with. Temporal coupling means that processes that are communicating will both have to be up and running. This coupling is analogous to the transient message-oriented communication we discussed in Chap. 4.

A different type of coordination occurs when processes are temporally decoupled, but referentially coupled, which we refer to as mailbox coordination. In this case, there is no need for two communicating processes to execute at the same time in order to let communication take place. Instead, communication takes place by putting messages in a (possibly shared) mailbox. This situation is analogous to persistent message-oriented communication as described in Chap. 4. It is necessary to explicitly address the mailbox that will hold the messages that are to be exchanged. Consequently, there is a referential coupling.

The combination of referentially decoupled and temporally coupled systems form the group of models for meeting-oriented coordination. In referentially decoupled systems, processes do not know each other explicitly. In other words, when a process wants to coordinate its activities with other processes, it cannot directly refer to another process. Instead, there is a concept of a meeting in which processes temporarily group together to coordinate their activities. The model prescribes that the meeting processes are executing at the same time.
[Page 591]

Meeting-based systems are often implemented by means of events, like the ones supported by object-based distributed systems. In this chapter, we discuss another mechanism for implementing meetings, namely publish/subscribe systems. In these systems, processes can subscribe to messages containing information on specific subjects, while other processes produce (i.e., publish) such messages. Most publish/subscribe systems require that communicating processes are active at the same time; hence there is a temporal coupling. However, the communicating processes may otherwise remain anonymous.

The most widely-known coordination model is the combination of referentially and temporally decoupled processes, exemplified by generative communication as introduced in the Linda programming system by Gelernter (1985). The key idea in generative communication is that a collection of independent processes make use of a shared persistent dataspace of tuples. Tuples are tagged data records consisting of a number (but possibly zero) typed fields. Processes can put any type of record into the shared dataspace (i.e., they generate communication records). Unlike the case with blackboards, there is no need to agree in advance on the structure of tuples. Only the tag is used to distinguish between tuples representing different kinds of information.

An interesting feature of these shared dataspaces is that they implement an associative search mechanism for tuples. In other words, when a process wants to extract a tuple from the dataspace, it essentially specifies (some of) the values of the fields it is interested in. Any tuple that matches that specification is then removed from the dataspace and passed to the process. If no match could be found, the process can choose to block until there is a matching tuple. We defer the details on this coordination model to later when discussing concrete systems.

We note that generative communication and shared dataspaces are often also considered to be forms of publish/subscribe systems. In what follows, we shall adopt this commonality as well. A good overview of publish/subscribe systems (and taking a rather broad perspective) can be found in Eugster et al. (2003). In this chapter we take the approach that in these systems there is at least referential decoupling between processes, but preferably also temporal decoupling.

13.2. Architectures

An important aspect of coordination-based systems is that communication takes place by describing the characteristics of data items that are to be exchanged. As a consequence, naming plays a crucial role. We return to naming later in this

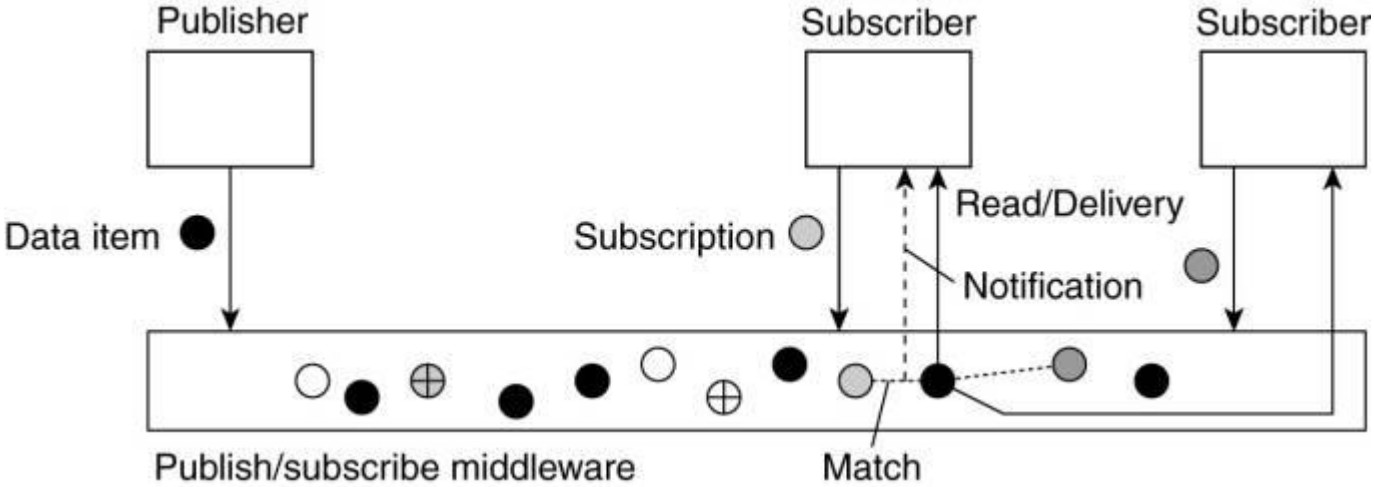
chapter, but for now the important issue is that in many cases, data items are not explicitly identified by senders and receivers.
[Page 592]

13.2.1. Overall Approach

Let us first assume that data items are described by a series of attributes. A data item is said to be published when it is made available for other processes to read. To that end, a subscription needs to be passed to the middleware, containing a description of the data items that the subscriber is interested in. Such a description typically consists of some (attribute, value) pairs, possibly combined with (attribute, range) pairs. In the latter case, the specified attribute is expected to take on values within a specified range. Descriptions can sometimes be given using all kinds of predicates formulated over the attributes, very similar in nature to SQL-like queries in the case of relational databases. We will come across these types of descriptors later in this chapter.

We are now confronted with a situation in which subscriptions need to be matched against data items, as shown in Fig. 13-2. When matching succeeds, there are two possible scenarios. In the first case, the middleware may decide to forward the published data to its current set of subscribers, that is, processes with a matching subscription. As an alternative, the middleware can also forward a notification at which point subscribers can execute a read operation to retrieve the published data item.

Figure 13-2. The principle of exchanging data items between publishers and subscribers.



In those cases in which data items are immediately forwarded to subscribers, the middleware will generally not offer storage of data. Storage is either explicitly handled by a separate service, or is the responsibility of subscribers. In other words, we have a referentially decoupled, but temporally coupled system.

This situation is different when notifications are sent so that subscribers need to explicitly read the published data. Necessarily, the middleware will have to store data items. In these situations there are additional operations for data management. It is also possible to attach a lease to a data item such that when the lease expires that the data item is automatically deleted.

In the model described so far, we have assumed that there is a fixed set of n attributes a1, . . . , an that is used to describe data items. In particular, each published data item is assumed to have an associated vector <(a1,v1),...,(an,vn)> of (attribute, value) pairs. In many coordination-based systems, this assumption is false. Instead, what happens is that events are published, which can be viewed as data items with only a single specified attribute.
[Page 593]

Events complicate the processing of subscriptions. To illustrate, consider a subscription such as "notify when room R4.20 is unoccupied and the door is unlocked." Typically, a distributed system supporting such subscriptions can be implemented by placing independent sensors for monitoring room occupancy (e.g., motion sensors) and those for registering the status of a door lock. Following the approach sketched so far, we would need to compose such primitive events into a publishable data item to which processes can then subscribe. Event composition turns out to be a difficult task, notably when the primitive events are generated from sources dispersed across the distributed system.

Clearly, in coordination-based systems such as these, the crucial issue is the efficient and scalable implementation of matching subscriptions to data items, along with the construction of relevant data items. From the outside, a coordination approach provides lots of potential for building very large-scale distributed systems due to the strong decoupling of processes. On the other hand, as we shall see next, devising scalable implementations without losing this independence is not a trivial exercise.

13.2.2. Traditional Architectures

The simplest solution for matching data items against subscriptions is to have a centralized client-server architecture. This is a typical solution currently adopted by many publish/subscribe systems, including IBM's WebSphere (IBM, 2005c)

and popular implementations for Sun's JMS (Sun Microsystems, 2004a). Likewise, implementations for the more elaborate generative communication models such as Jini (Sun Microsystems, 2005b) and JavaSpaces (Freeman et al., 1999) are mostly based on central servers. Let us take a look at two typical examples.

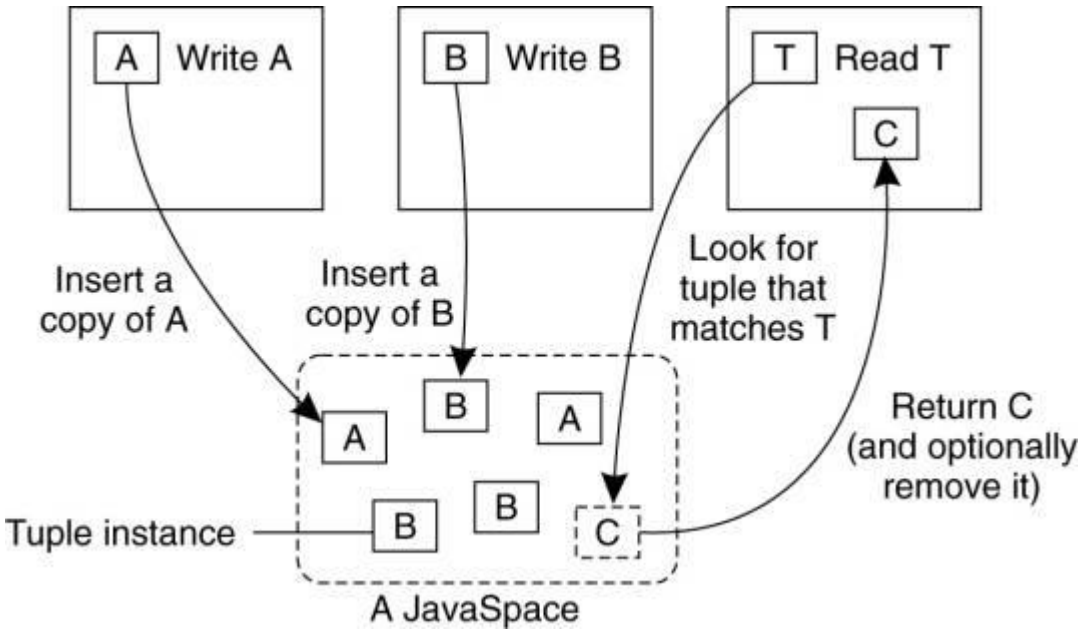
Example: Jini and JavaSpaces

Jini is a distributed system that consists of a mixture of different but related elements. It is strongly related to the Java programming language, although many of its principles can be implemented equally well in other languages. An important part of the system is formed by a coordination model for generative communication. Jini provides temporal and referential decoupling of processes through a coordination system called JavaSpaces (Freeman et al., 1999), derived from Linda. A JavaSpace is a shared dataspace that stores tuples representing a typed set of references to Java objects. Multiple JavaSpaces may coexist in a single Jini system.

Tuples are stored in serialized form. In other words, whenever a process wants to store a tuple, that tuple is first marshaled, implying that all its fields are marshaled as well. As a consequence, when a tuple contains two different fields that refer to the same object, the tuple as stored in a JavaSpace implementation will hold two marshaled copies of that object.
[Page 594]

A tuple is put into a JavaSpace by means of a write operation, which first marshals the tuple before storing it. Each time the write operation is called on a tuple, another marshaled copy of that tuple is stored in the JavaSpace, as shown in Fig. 13-3. We will refer to each marshaled copy as a tuple instance.

Figure 13-3. The general organization of a JavaSpace in Jini.



The interesting aspect of generative communication in Jini is the way that tuple instances are read from a JavaSpace. To read a tuple instance, a process provides another tuple that it uses as a template for matching tuple instances as stored in a JavaSpace. Like any other tuple, a template tuple is a typed set of object references. Only tuple instances of the same type as the template can be read from a JavaSpace. A field in the template tuple either contains a reference to an actual object or contains the value NULL. For example, consider the class

```
class public Tuple implements Entry {
  public Integer id, value;
    public Tuple(Integer id, Integer value){this.id = id; this.value = value}
}
```

Then a template declared as

Tuple template = new Tuple(null, new Integer(42))

will match the tuple

Tuple item = new Tuple("MyName", new Integer(42))

To match a tuple instance in a JavaSpace against a template tuple, the latter is marshaled as usual, including its NULL fields. For each tuple instance of the same type as the template, a field-by-field comparison is made with the template tuple. Two fields match if they both have a copy of the same reference or if the field in the template tuple is NULL. A tuple instance matches a template tuple if there is a pairwise matching of their respective fields.
[Page 595]

When a tuple instance is found that matches the template tuple provided as part of a read operation, that tuple instance is unmarshaled and returned to the reading process. There is also a take operation that additionally removes the tuple instance from the JavaSpace. Both operations block the caller until a matching tuple is found. It is possible to specify a maximum blocking time. In addition, there are variants that simply return immediately if no matching tuple existed.

Processes that make use of JavaSpaces need not coexist at the same time. In fact, if a JavaSpace is implemented using persistent storage, a complete Jini system can be brought down and later restarted without losing any tuples.

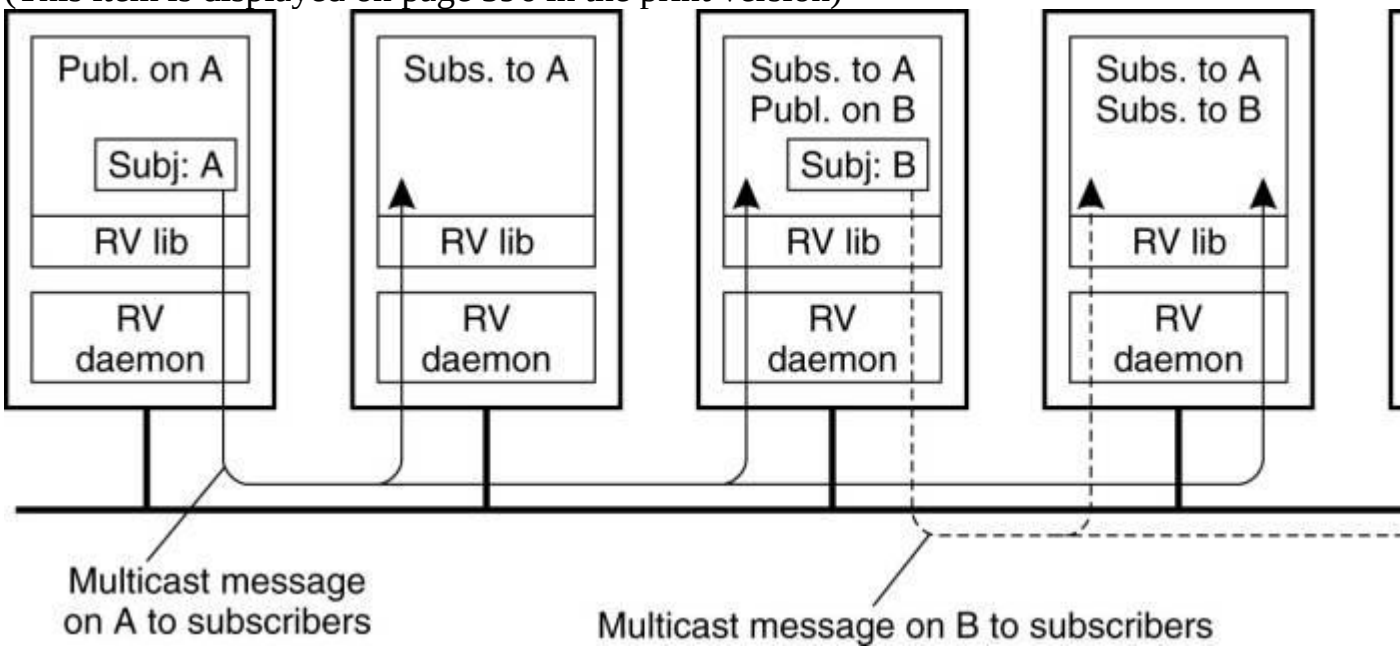
Although Jini does not support it, it should be clear that having a central server allows subscriptions to be fairly elaborate. For example, at the moment two nonnull fields match if they are identical. However, realizing that each field represents an object, matching could also be evaluated by executing an object-specific comparison operator [see also Picco et al. (2005)]. In fact, if such an operator can be overridden by an application, more-or-less arbitrary comparison semantics can be implemented. It is important to note that such comparisons may require an extensive search through currently stored data items. Such searches cannot be easily efficiently implemented in a distributed way. It is exactly for this reason that when elaborate matching rules are supported we will generally see only centralized implementations.

Another advantage of having a centralized implementation is that it becomes easier to implement synchronization primitives. For example, the fact that a process can block until a suitable data item is published, and then subsequently execute a destructive read by which the matching tuple is removed, offers facilities for process synchronization without processes needing to know each other. Again, synchronization in decentralized systems is inherently difficult as we also discussed in Chap. 6. We will return to synchronization below.

Example: TIB/Rendezvous

An alternative solution to using central servers is to immediately disseminate published data items to the appropriate subscribers using multicasting. This principle is used in TIB/Rendezvous, of which the basic architecture is shown in Fig. 13-4 (TIBCO, 2005) In this approach, a data item is a message tagged with a compound keyword describing its content, such as news.comp.os.books. A subscriber provides (parts of) a keyword, or indicating the messages it wants to receive, such as news.comp.*.books. These keywords are said to indicate the subject of a message.

Figure 13-4. The principle of a publish/subscribe system as implemented in TIB/Rendezvous.
(This item is displayed on page 596 in the print version)



Fundamental to its implementation is the use of broadcasting common in local-area networks, although it also uses more efficient communication facilities when possible. For example, if it is known exactly where a subscriber resides, point-to-point messages will generally be used. Each host on such a network will run a rendezvous daemon, which takes care that messages are sent and delivered according to their subject. Whenever a message is published, it is multicast to each host on the network running a rendezvous daemon. Typically, multicasting is implemented using the facilities offered by the underlying network, such as IP-multicasting or hardware broadcasting.
[Page 596]

Processes that subscribe to a subject pass their subscription to their local daemon. The daemon constructs a table of (process, subject), entries and whenever a

message on subject S arrives, the daemon simply checks in its table for local subscribers, and forwards the message to each one. If there are no subscribers for S, the message is discarded immediately.

When using multicasting as is done in TIB/Rendezvous, there is no reason why subscriptions cannot be elaborate and be more than string comparison as is currently the case. The crucial observation here is that because messages are forwarded to every node anyway, the potentially complex matching of published data against subscriptions can be done entirely locally without further network communication. However, as we shall discuss later, simple comparison rules are required whenever matching across wide-area networks is needed.

13.2.3. Peer-to-Peer Architectures

The traditional architectures followed by most coordination-based systems suffer from scalability problems (although their commercial vendors will state otherwise). Obviously, having a central server for matching subscriptions to published data cannot scale beyond a few hundred clients. Likewise, using multicasting requires special measures to extend beyond the realm of local-area networks. Moreover, if scalability is to be guaranteed, further restrictions on describing subscriptions and data items may be necessary.

Much research has been spent on realizing coordination-based systems using peer-to-peer technology. Straightforward implementations exist for those cases in which keywords are used, as these can be hashed to unique identifiers for published data. This approach has also been used for mapping (attribute, value) pairs to identifiers. In these cases, matching reduces to a straightforward lookup of an identifier, which can be efficiently implemented in a DHT-based system. This approach works well for the more conventional publish/subscribe systems as illustrated by Tam and Jacobsen (2003), but also for generative communication (Busi et al., 2004).

Matters become complicated for more elaborate matching schemes. Notoriously difficult are the cases in which ranges need to be supported and only very few proposals exist. In the following, we discuss one such proposal, devised by one of the authors and his colleagues (Voulgaris et al., 2006).

Example: A Gossip-Based Publish/Subscribe System

Consider a publish/subscribe system in which data items can be described by means of N attributes a1, . . . , aN whose value can be directly mapped to a floating-point number. Such values include, for example, floats, integers,

enumerations, booleans, and strings. A subscription s takes the form of a tuple of (attribute, value/range) pairs, such as

$s = \langle a1 \rightarrow 3.0, a4 \rightarrow [0.0, 0.5) \rangle$

In this example, s specifies that a1 should be equal to 3.0, and a4 should lie in the interval [0.0, 0.5). Other attributes are allowed to take on any value. For clarity, assume that every node i enters only one subscription si.

Note that each subscription si actually specifies a subset Si in the N-dimensional space of floating-point numbers. Such a subset is also called a hyperspace. For the system as a whole, only published data whose description falls in the union $S = \bigcup Si$ of these hyperspaces is of interest. The whole idea is to automatically partition S into M disjoint hyperspaces S1, . . . , SM such that each falls completely in one of the subscription hyperspaces Si, and together they cover all subscriptions. More formally, we have that:

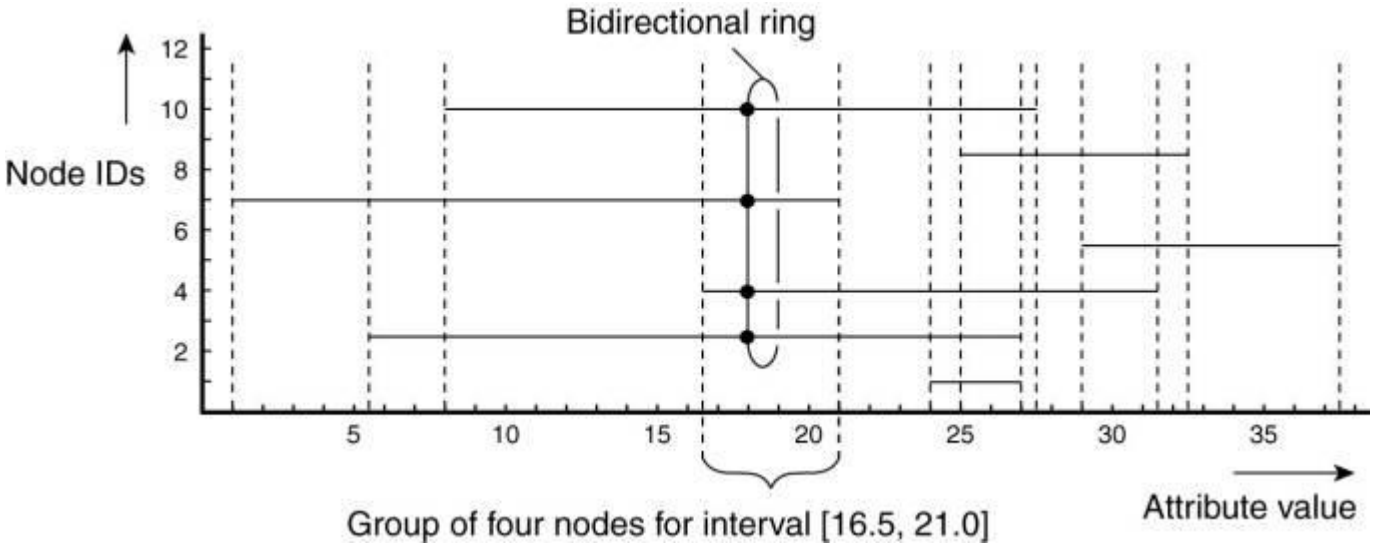
$(Sm \cap Si \neq \emptyset) \Rightarrow (Sm \subseteq Si)$

Moreover, the system keeps M minimal in the sense that there is no partitioning with fewer parts Sm. The whole idea is to register, for each hyperspace Sm, exactly those nodes i for which $Sm \subseteq Si$. In that case, when a data item is published, the system need merely find the Sm to which that item belongs, from which point it can forward the item to the associated nodes.
[Page 598]

To this end, nodes regularly exchange subscriptions using an epidemic protocol. If two nodes i and j notice that their respective subscriptions intersect, that is $Sij \equiv Si \cap Sj \neq \emptyset$ they will record this fact and keep references to each other. If they discover a third node k with $Sijk \equiv Sij \cap Sk \neq \emptyset$, the three of them will connect to each other so that a data item d from Sijk can be efficiently disseminated. Note that if $Sij - Sijk \neq \emptyset$, nodes i and j will maintain their mutual references, but now associate it strictly with Sij - Sijk.

In essence, what we are seeking is a means to cluster nodes into M different groups, such that nodes i and j belong to the same group if and only if their subscriptions Si and Sj intersect. Moreover, nodes in the same group should be organized into an overlay network that will allow efficient dissemination of a data item in the hyperspace associated with that group. This situation for a single attribute is sketched in Fig. 13-5.

Figure 13-5. Grouping nodes for supporting range queries in a peer-to-peer publish/subscribe system.



Here, we see a total of seven nodes in which the horizontal line for node i indicates its range of interest for the value of the single attribute. Also shown is the grouping of nodes into disjoint ranges of interests for values of the attribute. For example, nodes 3, 4, 7, and 10 will be grouped together representing the interval [16.5, 21.0]. Any data item with a value in this range should be disseminated to only these four nodes.

To construct these groups, the nodes are organized into a gossip-based unstructured network. Each node maintains a list of references to other neighbors (i.e., a partial view), which it periodically exchanges with one of its neighbors as described in Chap. 2. Such an exchange will allow a node to learn about random other nodes in the system. Every node keeps track of the nodes it discovers with overlapping interests (i.e., with an intersecting subscription).

At a certain moment, every node i will generally have references to other nodes with overlapping interests. As part of exchanging information with a node j, node i orders these nodes by their identifiers and selects the one with the lowest identifier $i1 > j$, such that its subscription overlaps with that of node j, that is, $S_{j,i1} \equiv S_{i1} \cap S_j \neq \emptyset$.
[Page 599]

The next one to be selected is $i2 > i1$ such that its subscription also overlaps with that of j, but only if it contains elements not yet covered by node i1. In other words, we should have that $S_{j,i1,i2} \equiv (S_{i2} - S_{j,i1}) \cap S_j \neq \emptyset$. This process is repeated until all nodes that have an overlapping interest with node i have been inspected, leading to an ordered list $i1 < i2 < \ldots < in$. Note that a node ik is in

this list because it covers a region R of common interest to node i and j not yet jointly covered by nodes with a lower identifier than ik. In effect, node ik is the first node that node j should forward a data item to that falls in this unique region R. This procedure can be expanded to let node i construct a bidirectional ring. Such a ring is also shown in Fig. 13-5.

Whenever a data item d is published, it is disseminated as quickly as possible to any node that is interested in it. As it turns out, with the information available at every node finding a node i interested in d is simple. From there on, node i need simply forward d along the ring of subscribers for the particular range that d falls into. To speed up dissemination, short-cuts are maintained for each ring as well. Details can be found in Voulgaris et al. (2006).

Discussion

An approach somewhat similar to this gossip-based solution in the sense that it attempts to find a partitioning of the space covered by the attribute's values, but which uses a DHT-based system is described in Gupta et al. (2004). In another proposal described in Bharambe (2004), each attribute ai is handled by a separate process Pi, which in turn partitions the range of its attribute across multiple processes. When a data item d is published, it is forwarded to each Pi, where it is subsequently stored at the process responsible for the d's value of ai.

All these approaches are illustrative for the complexity when mapping a nontrivial publish/subscribe system to a peer-to-peer network. In essence, this complexity comes from the fact that supporting search in attribute-based naming systems is inherently difficult to establish in a decentralized fashion. We will again come across these difficulties when discussing replication.

13.2.4. Mobility and Coordination

A topic that has received considerable attention in the literature is how to combine publish/subscribe solutions with node mobility. In many cases, it is assumed that there is a fixed basic infrastructure with access points for mobile nodes. Under these assumptions, the issue becomes how to ensure that published messages are not delivered more than once to a subscriber who switches access points. One practical solution to this problem is to let subscribers keep track of the messages they have already received and simply discard duplicates. Alternative, but more intricate solutions comprise routers that keep track of which messages have been sent to which subscribers (see, e.g., Caporuscio et al., 2003).

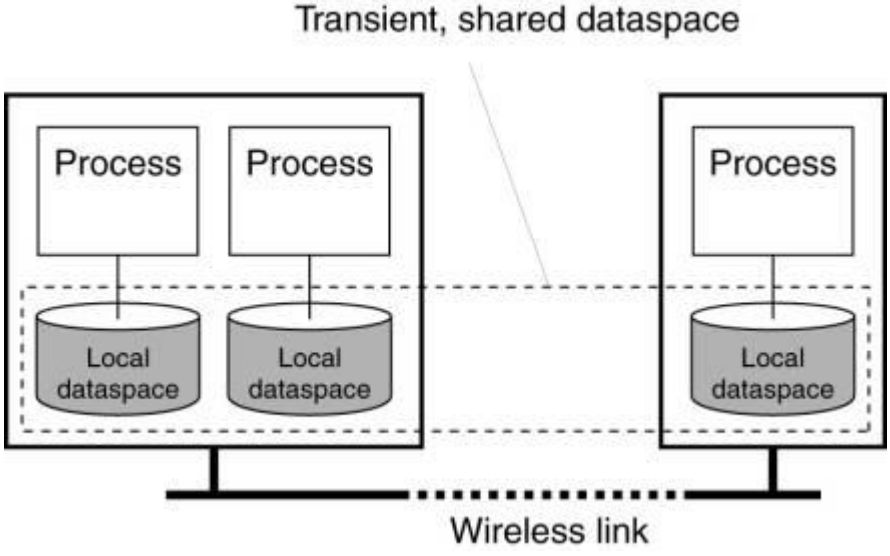
Example: Lime

In the case of generative communication, several solutions have been proposed to operate a shared dataspace in which (some of) the nodes are mobile. A canonical example in this case is Lime (Murphy et al., 2001), which strongly resembles the JavaSpace model we discussed previously.

In Lime, each process has its own associated dataspace, but when processes are in each other's proximity such that they are connected, their dataspaces become shared. Theoretically, being connected can mean that there is a route in a joint underlying network that allows two processes to exchange data. In practice, however, it either means that two processes are temporarily located on the same physical host, or their respective hosts can communicate with each other through a (single hop) wireless link. Formally, the processes should be member of the same group and use the same group communication protocol.

The local dataspaces of connected processes form a transiently shared dataspace that will allow processes to exchange tuples, as shown in Fig. 13-6. For example, when a process P executes a write operation, the associated tuple is stored in the process's local dataspace. In principle, it stays there until there is a matching take operation, possibly from another process that is now in the same group as P. In this way, the fact that we are actually dealing with a completely distributed shared dataspace is transparent for participating processes. However, Lime also allows breaking this transparency by specifying exactly for whom a tuple is intended. Likewise, read and take operations can have an additional parameter specifying from which process a tuple is expected.

Figure 13-6. Transient sharing of local dataspaces in Lime.



To better control how tuples are distributed, dataspaces can carry out what are known as reactions. A reaction specifies an action to be executed when a tuple matching a given template is found in the local dataspace. Each time a dataspace changes, an executable reaction is selected at random, often leading to a further modification of the dataspace. Reactions span the current shared dataspace, but there are several restrictions to ensure that they can be executed efficiently. For example, in the case of weak reactions, it is only guaranteed that the associated actions are eventually executed, provided the matching data is still accessible.
[Page 601]

The idea of reactions has been taken a step further in TOTA, where each tuple has an associated code fragment telling exactly how that tuple should be moved between dataspaces, possibly also including transformations (Mamei and Zambonelli, 2004).

13.3. Processes

There is nothing really special about the processes used in publish/subscribe systems. In most cases, efficient mechanisms need to be deployed for searching in a potentially large collection of data. The main problem is devising schemes that work well in distributed environments. We return to this issue below when discussing consistency and replication.

13.4. Communication

Communication in many publish/subscribe systems is relatively simple. For example, in virtually every Java-based system, all communication proceeds through remote method invocations. One important problem that needs to be handled when publish/subscribe systems are spread across a wide-area system is that published data should reach only the relevant subscribers. As we described above, using a self-organizing method by which nodes in a peer-to-peer system are automatically clustered, after which dissemination takes place per cluster is one solution. An alternative solution is to deploy content-based routing.

13.4.1. Content-Based Routing

In content-based routing, the system is assumed to be built on top of a point-to-point network in which messages are explicitly routed between nodes. Crucial in this setup is that routers can take routing decisions by considering the content of

a message. More precisely, it is assumed that each message carries a description of its content, and that this description can be used to cut-off routes for which it is known that they do not lead to receivers interested in that message.

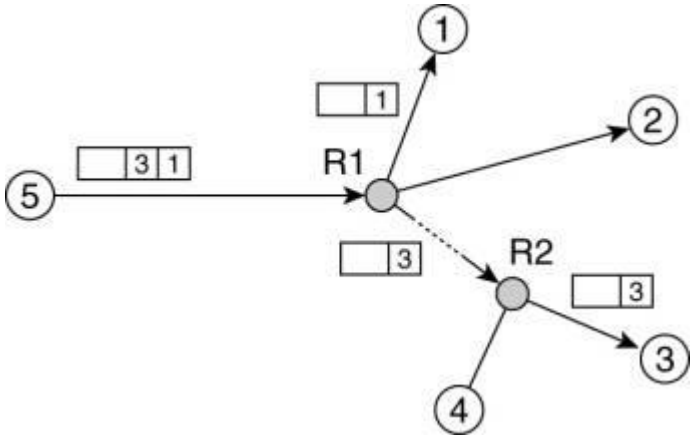
A practical approach toward content-based routing is proposed in Carzaniga et al. (2004). Consider a publish/subscribe system consisting of N servers to which clients (i.e., applications) can send messages, or from which they can read incoming messages. We assume that in order to read messages, an application will have previously provided the server with a description of the kind of data it is interested in. The server, in turn, will notify the application when relevant data has arrived.
[Page 602]

Carzaniga et al. propose a two-layered routing scheme in which the lowest layer consists of a shared broadcast tree connecting the N servers. There are various ways for setting up such a tree, ranging from network-level multicast support to application-level multicast trees as we discussed in Chap. 4. Here, we also assume that such a tree has been set up with the N servers as end nodes, along with a collection of intermediate nodes forming the routers. Note that the distinction between a server and a router is only a logical one: a single machine may host both kinds of processes.

Consider first two extremes for content-based routing, assuming we need to support only simple subject-based publish/subscribe in which each message is tagged with a unique (noncompound) keyword. One extreme solution is to send each published message to every server, and subsequently let the server check whether any of its clients had subscribed to the subject of that message. In essence, this is the approach followed in TIB/Rendezvous.

The other extreme solution is to let every server broadcast its subscriptions to all other servers. As a result, every server will be able to compile a list of (subject, destination) pairs. Then, whenever an application submits a message on subject s, its associated server prepends the destination servers to that message. When the message reaches a router, the latter can use the list to decide on the paths that the message should follow, as shown in Fig. 13-7.

Figure 13-7. Naive content-based routing.



Taking this last approach as our starting point, we can refine the capabilities of routers for deciding where to forward messages to. To that end, each server broadcasts its subscription across the network so that routers can compose routing filters. For example, assume that node 3 in Fig. 13-7 subscribes to messages for which an attribute a lies in the range [0,3], but that node 4 wants messages with a ε [2,5]. In this case, router R2 will create a routing filter as a table with an entry for each of its outgoing links (in this case three: one to node 3, one to node 4, and one toward router R1), as shown in Fig. 13-8.
[Page 603]

Figure 13-8. A partially filled routing table.

Interface	Filter
To node 3	a ε [0,3]
To node 4	a ε [2,5]
Toward router R1	(unspecified)

More interesting is what happens at router R1. In this example, the subscriptions from nodes 3 and 4 dictate that any message with a lying in the interval [0,3] ∪ [2,5] = [0,5] should be forwarded along the path to router R2, and this is precisely the information that R1 will store in its table. It is not difficult to imagine that more intricate subscription compositions can be supported.

This simple example also illustrates that whenever a node leaves the system, or when it is no longer interested in specific messages, it should cancel its subscription and essentially broadcast this information to all routers. This cancellation, in turn, may lead to adjusting various routing filters. Late adjustments will at worst lead to unnecessary traffic as messages may be

forwarded along paths for which there are no longer subscribers. Nevertheless, timely adjustments are needed to keep performance at an acceptable level.

One of the problems with content-based routing is that although the principle of composing routing filters is simple, identifying the links along which an incoming message must be forwarded can be compute-intensive. The computational complexity comes from the implementation of matching attribute values to subscriptions, which essentially boils down to an entry-by-entry comparison. How this comparison can be done efficiently is described in Carzaniga et al. (2003).

13.4.2. Supporting Composite Subscriptions

The examples so far form relatively simple extensions to routing tables. These extensions suffice when subscriptions take the form of vectors of (attribute, value/range) pairs. However, there is often a need for more sophisticated expressions of subscriptions. For example, it may be convenient to express compositions of subscriptions in which a process specifies in a single subscription that it is interested in very different types of data items. To illustrate, a process may want to see data items on stocks from IBM and data on their revenues, but sending data items of only one kind is not useful.

To handle subscription compositions, Li and Jacobsen (2005) proposed to design routers analogous to rule databases. In effect, subscriptions are transformed into rules stating under which conditions published data should be forwarded, and along which outgoing links. It is not difficult to imagine that this may lead to content-based routing schemes that are far more advanced than the routing filters described above. Supporting subscription composition is strongly related to naming issues in coordination-based systems, which we discuss next.

13.5. Naming

Let us now pay some more attention to naming in coordination-based systems. So far, we have mostly assumed that every published data item has an associated vector of n (attribute, value) pairs and that processes can subscribe to data items by specifying predicates over these attribute values. In general, this naming scheme can be readily applied, although systems differ with respect to attribute types, values, and the predicates that can be used.

For example, with JavaSpaces we saw that essentially only comparison for equality is supported, although this can be relatively easily extended in

application-specific ways. Likewise, many commercial publish/subscribe systems support only rather primitive string-comparison operators.

One of the problems we already mentioned is that in many cases we cannot simply assume that every data item is tagged with values for all attributes. In particular, we will see that a data item has only one associated (attribute, value) pair, in which case it is also referred to as an event. Support for subscribing to events, and notably composite events largely dictates the discussion on naming issues in publish/subscribe systems. What we have discussed so far should be considered as the more primitive means for supporting coordination in distributed systems. We now address in more depth events and event composition.

When dealing with composite events, we need to take two different issues into account. The first one is to describe compositions. Such descriptions form the basis for subscriptions. The second issue is how to collect (primitive) events and subsequently match them to subscriptions. Pietzuch et al. (2003) have proposed a general framework for event composition in distributed systems. We take this framework as the basis for our discussion.

13.5.1. Describing Composite Events

Let us first consider some examples of composite events to give a better idea of the complexity that we may need to deal with. Fig. 13-9 shows examples of increasingly complex composite events. In this example, R4.20 could be an air-conditioned and secured computer room.

Figure 13-9. Examples of events in a distributed system.
(This item is displayed on page 605 in the print version)

Ex.	Description
S1	Notify when room R4.20 is unoccupied
S2	Notify when R4.20 is unoccupied and the door is unlocked
S3	Notify when R4.20 is unoccupied for 10 seconds while the door is unlocked
S4	Notify when the temperature in R4.20 rises more than 1 degree per 30 minutes
S5	Notify when the average temperature in R4.20 is more than 20 degrees in the past 30 minutes

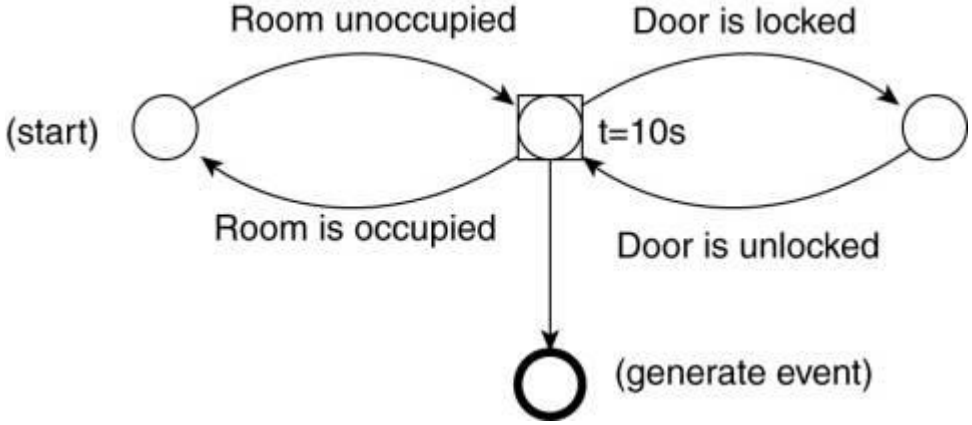
The first two subscriptions are relatively easy. S1 is an example that can be handled by a primitive discrete event, whereas S2 is a simple composition of two discrete events. Subscription S3 is more complex as it requires that the system can also report time-related events. Matters are further complicated if

subscriptions involve aggregated values required for computing gradients (S4) or averages (S5). Note that in the case of S5 we are requiring a continuous monitoring of the system in order to send notifications on time.
[Page 605]

The basic idea behind an event-composition language for distributed systems is to enable the formulation of subscriptions in terms of primitive events. In their framework, Pietzuch et al. provide a relatively simple language for an extended type of finite-state machine (FSM). The extensions allow for the specification of sojourn times in states, as well as the generation of new (composite) events. The precise details of their language are not important for our discussion here. What is important is that subscriptions can be translated into FSMs.

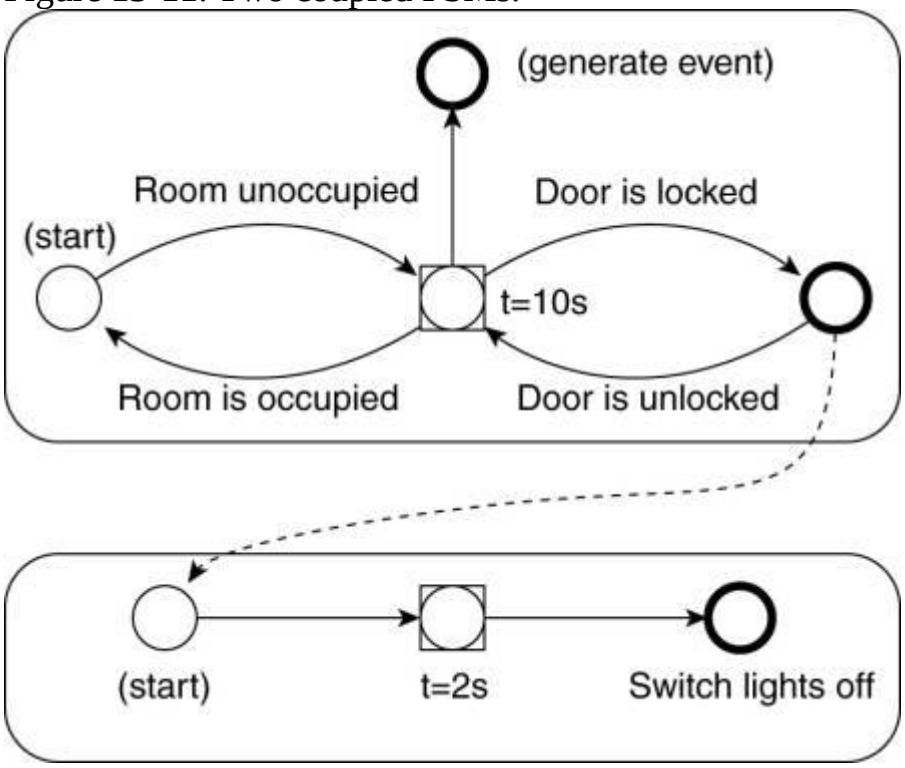
To give an example, Fig. 13-10 shows the FSM for subscription S3 from Fig. 13-9. The special case is given by the timed state, indicated by the label "t = 10s" which specifies that a transition to the final state is made if the door is not locked within 10 seconds.

Figure 13-10. The finite state machine for subscription S3 from Fig. 13-9.



Much more complex subscriptions can be described. An important aspect is that these FSMs can often be decomposed into smaller FSMs that communicate by passing events to each other. Note that such an event communication would normally trigger a state transition at the FSM for which that event is intended. For example, assume that we want to automatically turn off the lights in room R4.20 after 2 seconds when we are certain that nobody is there anymore (and the door is locked). In that case, we can reuse the FSM from Fig. 13-10 if we let it generate an event for a second FSM that will trigger the lighting, as shown in Fig. 13-11
[Page 606]

Figure 13-11. Two coupled FSMs.



The important observation here is that these two FSMs can be implemented as separate processes in the distributed system. In this case, the FSM for controlling the lighting will subscribe to the composed event that is triggered when R4.20 is unoccupied and the door is locked. This leads to distributed detectors which we discuss next.

13.5.2. Matching Events and Subscriptions

Now consider a publish/subscribe system supporting composite events. Every subscription is provided in the form of an expression that can be translated into a finite state machine (FSM). State transitions are essentially triggered by primitive events that take place, such as leaving a room or locking a door.

To match events and subscriptions, we can follow a simple, naive implementation in which every subscriber runs a process implementing the finite state machine associated with its subscription. In that case, all the primitive events that are relevant for a specific subscription will have to be forwarded to the subscriber. Obviously, this will generally not be very efficient.

A much better approach is to consider the complete collection of subscriptions, and decompose subscriptions into communicating finite state machines, such that some of these FSMs are shared between different subscriptions. An example of this sharing was shown in Fig. 13-11. This approach toward handling subscriptions leads to what are known as distributed event detectors. Note that a distribution of event detectors is similar in nature to the distributed resolution of names in various naming systems. Primitive events lead to state transitions in relatively simple finite state machines, in turn triggering the generation of composite events. The latter can then lead to state transitions in other FSMs, again possibly leading to further event generation. Of course, events translate to messages that are sent over the network to processes that subscribed to them.
[Page 607]

Besides optimizing through sharing, breaking down subscriptions into communicating FSMs also has the potential advantage of optimizing network usage. Consider again the events related to monitoring the computer room we described above. Assuming that there only processes interested in the composite events, it makes sense to compose these events close to the computer room. Such a placement will prevent having to send the primitive events across the network. Moreover, when considering Fig. 13-9, we see that we may only need to send the alarm when noticing that the room is unoccupied for 10 seconds while the door is unlocked. Such an event will generally occur rarely in comparison to, for example, (un)locking the door.

Decomposing subscriptions into distributed event detectors, and subsequently optimally placing them across a distributed system is still subject to much research. For example, the last word on subscription languages has not been said, and especially the trade-off between expressiveness and efficiency of implementations will attract a lot of attention. In most cases, the more expressive a language is, the more unlikely there will be an efficient distributed implementation. Current proposals such as by Demers et al. (2006) and by Liu and Jacobsen (2004) confirm this. It will take some years before we see these techniques being applied to commercial publish/subscribe systems.

13.6. Synchronization

Synchronization in coordination-based systems is generally restricted to systems supporting generative communication. Matters are relatively straightforward when only a single server is used. In that case, processes can be simply blocked until tuples become available, but it is also simpler to remove them. Matters

become complicated when the shared dataspace is replicated and distributed across multiple servers, as we describe next.

13.7. Consistency and Replication

Replication plays a key role in the scalability of coordination-based systems, and notably those for generative communication. In the following, we first consider some standard approaches as have been explored in a number of systems such as JavaSpaces. Next, we describe some recent results that allow for the dynamic and automatic placement of tuples depending on their access patterns.
[Page 608]

13.7.1. Static Approaches

The distributed implementation of a system supporting generative communication frequently requires special attention. We concentrate on possible distributed implementations of a JavaSpace server, that is, an implementation by which the collection of tuple instances may be distributed and replicated across several machines. An overview of implementation techniques for tuple-based runtime systems is given by Rowstron (2001).

1. General Considerations

2. An efficient distributed implementation of a JavaSpace has to solve two problems:

3. How to simulate associative addressing without massive searching.

4. How to distribute tuple instances among machines and locate them later.

The key to both problems is to observe that each tuple is a typed data structure. Splitting the tuple space into subspaces, each of whose tuples is of the same type simplifies programming and makes certain optimizations possible. For example, because tuples are typed, it becomes possible to determine at compile time which subspace a call to a write, read, or take operates on. This partitioning means that only a fraction of the set of tuple instances has to be searched.

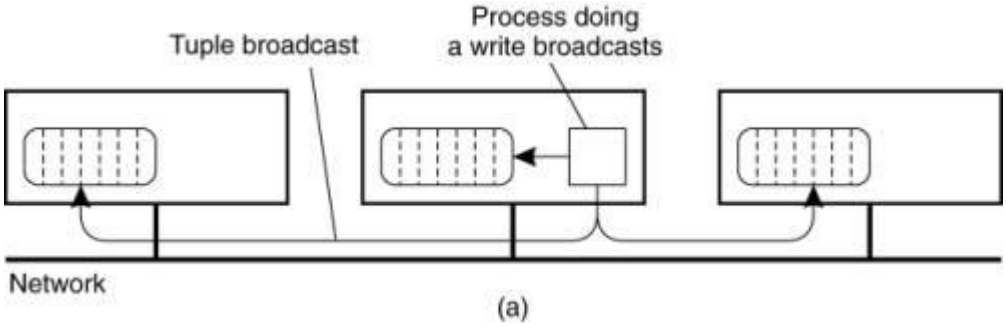
In addition, each subspace can be organized as a hash table using (part of) its i-th tuple field as the hash key. Recall that every field in a tuple instance is a marshaled reference to an object. JavaSpaces does not prescribe how marshaling should be done. Therefore, an implementation may decide to marshal a reference

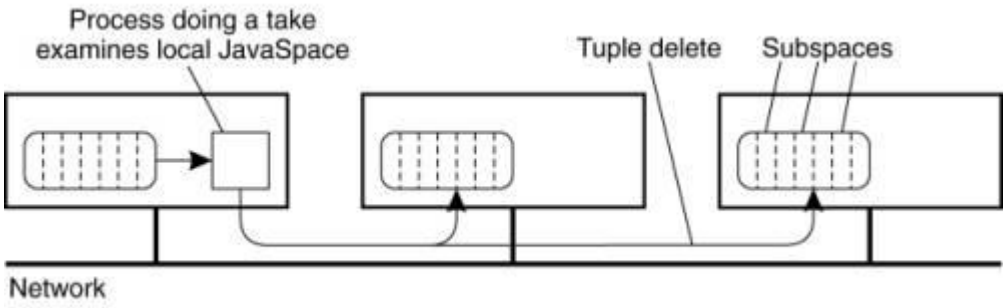
in such a way that the first few bytes are used as an identifier of the type of the object that is being marshaled. A call to a write, read, or take operation can then be executed by computing the hash function of the ith field to find the position in the table where the tuple instance belongs. Knowing the subspace and table position eliminates all searching. Of course, if the ith field of a read or take operation is NULL, hashing is not possible, so a complete search of the subspace is generally needed. By carefully choosing the field to hash on, however, searching can often be avoided.

Additional optimizations are also used. For example, the hashing scheme described above distributes the tuples of a given subspace into bins to restrict searching to a single bin. It is possible to place different bins on different machines, both to spread the load more widely and to take advantage of locality. If the hashing function is the type identifier modulo the number of machines, the number of bins scales linearly with the system size [see also Bjornson (1993)].
[Page 609]

On a network of computers, the best choice depends on the communication architecture. If reliable broadcasting is available, a serious candidate is to replicate all the subspaces in full on all machines, as shown in Fig. 13-12. When a write is done, the new tuple instance is broadcast and entered into the appropriate subspace on each machine. To do a read or take operation, the local subspace is searched. However, since successful completion of a take requires removing the tuple instance from the JavaSpace, a delete protocol is required to remove it from all machines. To prevent race conditions and deadlocks, a two-phase commit protocol can be used.

Figure 13-12. A JavaSpace can be replicated on all machines. The dotted lines show the partitioning of the JavaSpace into subspaces. (a) Tuples are broadcast on write. (b) reads are local, but the removing an instance when calling take must be broadcast.



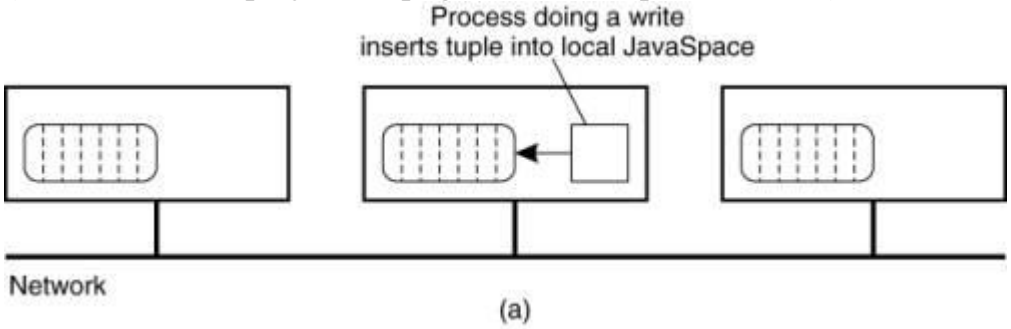


(b)

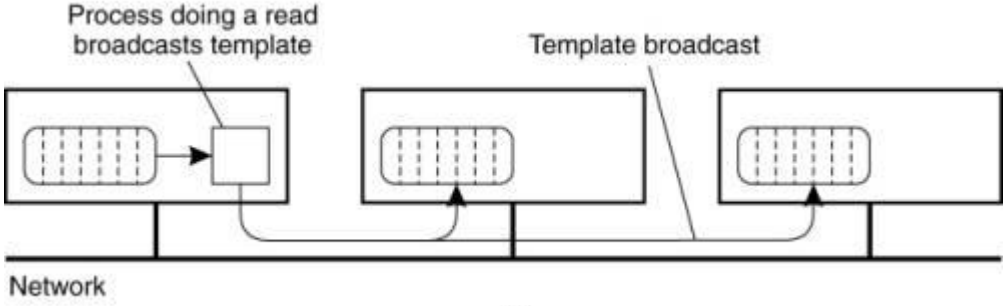
This design is straightforward, but may not scale well as the system grows in the number of tuple instances and the size of the network. For example, implementing this scheme across a wide-area network is prohibitively expensive.

The inverse design is to do writes locally, storing the tuple instance only on the machine that generated it, as shown in Fig. 13-13. To do a read or take, a process must broadcast the template tuple. Each recipient then checks to see if it has a match, sending back a reply if it does.

Figure 13-13. Nonreplicated JavaSpace. (a) A write is done locally. (b) A read or take requires the template tuple to be broadcast in order to find a tuple instance. (This item is displayed on page 610 in the print version)



(a)

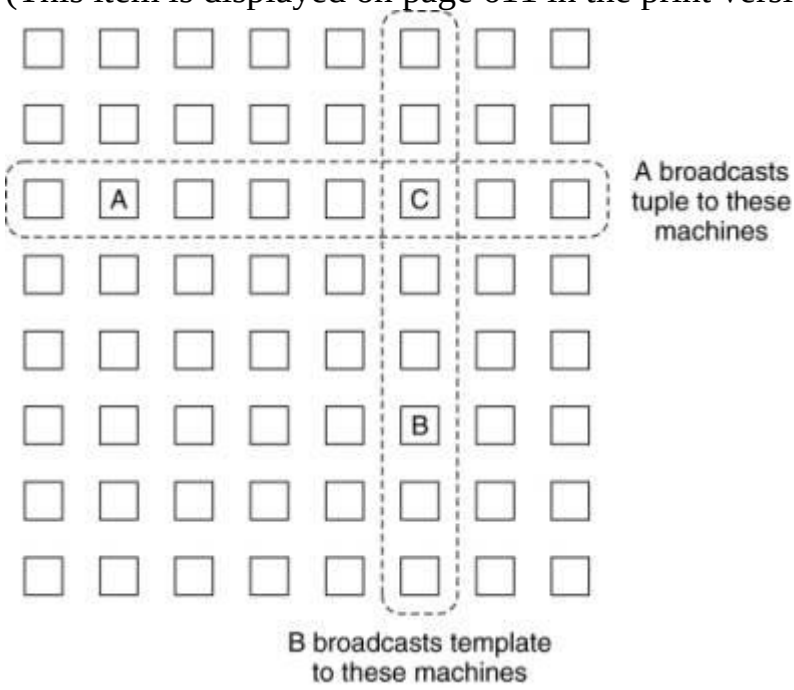


(b)

If the tuple instance is not present, or if the broadcast is not received at the machine holding the tuple, the requesting machine retransmits the broadcast request ad infinitum, increasing the interval between broadcasts until a suitable tuple instance materializes and the request can be satisfied. If two or more tuple instances are sent, they are treated like local writes and the instances are effectively moved from the machines that had them to the one doing the request. In fact, the runtime system can even move tuples around on its own to balance the load. Carriero and Gelernter (1986) used this method for implementing the Linda tuple space on a LAN.
[Page 610]

These two methods can be combined to produce a system with partial replication. As a simple example, imagine that all the machines logically form a rectangular grid, as shown in Fig. 13-14. When a process on a machine A wants to do a write, it broadcasts (or sends by point-to-point message) the tuple to all machines in its row of the grid. When a process on a machine B wants to read or take a tuple instance, it broadcasts the template tuple to all machines in its column. Due to the geometry, there will always be exactly one machine that sees both the tuple instance and the template tuple (C in this example), and that machine makes the match and sends the tuple instance to the process requesting for it. This approach is similar to using quorum-based replication as we discussed in Chap. 7.

Figure 13-14. Partial broadcasting of tuples and template tuples.
(This item is displayed on page 611 in the print version)



The implementations we have discussed so far have serious scalability problems caused by the fact that multicasting is needed either to insert a tuple into a tuple space, or to remove one. Wide-area implementations of tuple spaces do not exist. At best, several different tuple spaces can coexist in a single system, where each tuple space itself is implemented on a single server or on a local-area network. This approach is used, for example, in PageSpaces (Ciancarini et al., 1998) and WCL (Rowstron and Wray, 1998). In WCL, each tuple-space server is responsible for an entire tuple space. In other words, a process will always be directed to exactly one server. However, it is possible to migrate a tuple space to a different server to enhance performance. How to develop an efficient wide-area implementation of tuple spaces is still an open question.
[Page 611]

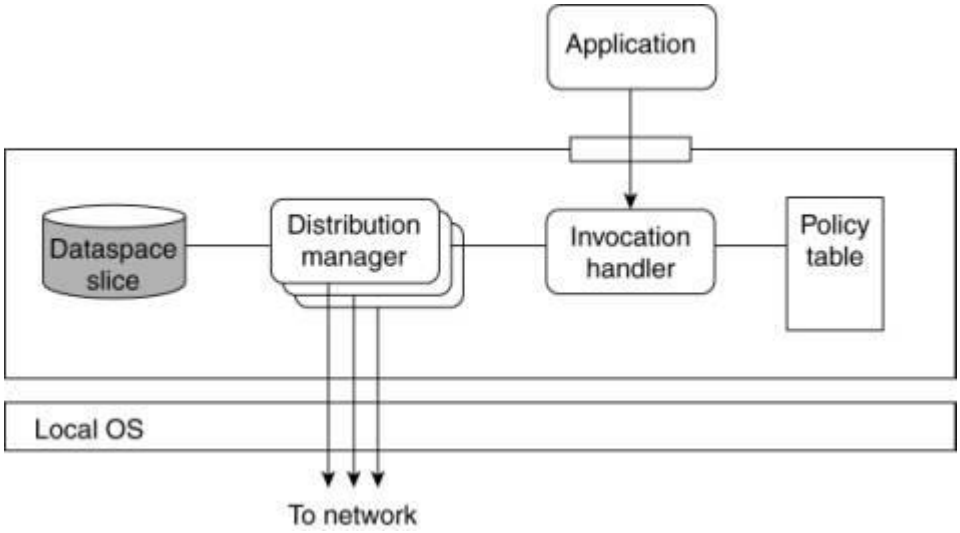
13.7.2. Dynamic Replication

Replication in coordination-based systems has generally been restricted to static policies for parallel applications like those discussed above. In commercial applications, we also see relatively simple schemes in which entire dataspaces or otherwise statically predefined parts of a data set are subject to a single policy (GigaSpaces, 2005). Inspired by the fine-grained replication of Web documents in Globule, performance improvements can also be achieved when differentiating replication between the different kinds of data stored in a dataspace. This differentiation is supported by GSpace, which we briefly discuss in this section.

GSpace Overview

GSpace is a distributed coordination-based system that is built on top of JavaSpaces (Russello et al., 2004, 2006). Distribution and replication of tuples in GSpace is done for two different reasons: improving performance and availability. A key element in this approach is the separation of concerns: tuples that need to be replicated for availability may need to follow a different strategy than those for which performance is at stake. For this reason, the architecture of GSpace has been set up to support a variety of replication policies, and such that different tuples may follow different policies.
[Page 612]

Figure 13-15. Internal organization of a GSpace kernel.



The principal working is relatively simple. Every application is offered an interface with a read, write, and take interface, similar to what is offered by JavaSpaces. However, every call is picked up by a local invocation handler which looks up the policy that should be followed for the specific call. A policy is selected based on the type and content of the tuple/template that is passed as part of the call. Every policy is identified by a template, similar to the way that templates are used to select tuples in other Java-based shared dataspaces as we discussed previously.

The result of this selection is a reference to a distribution manager, which implements the same interface, but now does it according to a specific replication policy. For example, if a master/slave policy has been implemented, a read operation may be implemented by immediately reading a tuple from the locally available dataspace. Likewise, a write operation may require that the distribution manager forwards the update to the master node and awaits an acknowledgment before performing the operation locally.

Finally, every GSpace kernel has a local dataspace, called a slice, which is implemented as a full-fledged, nondistributed version of JavaSpaces.

In this architecture (of which some components are not shown for clarity), policy descriptors can be added at runtime, and likewise, distribution managers can be changed as well. This setup allows for a fine-grained tuning of the distribution and replication of tuples, and as is shown in Russello et al. (2004), such fine-tuning allows for much higher performance than is achievable with any fixed, global strategy that is applied to all tuples in a dataspace.
[Page 613]

Adaptive Replication

However, the most important aspect with systems such as GSpace is that replication management is automated. In other words, rather than letting the application developer figure out which combination of policies is the best, it is better to let the system monitor access patterns and behavior and subsequently adopt policies as necessary.

To this end, GSpace follows the same approach as in Globule: it continuously measures consumed network bandwidth, latency, and memory usage and depending on which of these metrics is considered most important, places tuples on different nodes and chooses the most appropriate way to keep replicas consistent. The evaluation of which policy is the best for a given tuple is done by means of a central coordinator which simply collects traces from the nodes that constitute the GSpace system.

An interesting aspect is that from time to time we may need to switch from one replication policy to another. There are several ways in which such a transition can take place. As GSpace aims to separate mechanisms from policies as best as possible, it can also handle different transition policies. The default case is to temporarily freeze all operations for a specific type of tuple, remove all replicas and reinsert the tuple into the shared dataspace but now following the newly selected replication policy. However, depending on the new replication policy, a different way of making the transition may be possible (and cheaper). For example, when switching from no replication to master/slave replication, one approach could be to lazily copy tuples to the slaves when they are first accessed.

13.8. Fault Tolerance

When considering that fault tolerance is fundamental to any distributed system, it is somewhat surprising how relatively little attention has been paid to fault tolerance in coordination-based systems, including basic publish/subscribe systems as well as those supporting generative communication. In most cases, attention focuses on ensuring efficient reliability of data delivery, which essentially boils down to guaranteeing reliable communication. When the middleware is also expected to store data items, as is the case with generative communication, some effort is paid to reliable storage. Let us take a closer look at these two cases.

13.8.1. Reliable Publish-Subscribe Communication

In coordination-based systems where published data items are matched only against live subscribers, reliable communication plays a crucial role. In this case, fault tolerance is most often implemented through the implementation of reliable multicast systems that underly the actual publish/subscribe software. There are several issues that are generally taken care of. First, independent of the way that content-based routing takes place, a reliable multicast channel is set up. Second, process fault tolerance needs to be handled. Let us take a look how these matters are addressed in TIB/Rendezvous.
[Page 614]

Example: Fault Tolerance in TIB/Rendezvous

TIB/Rendezvous assumes that the communication facilities of the underlying network are inherently unreliable. To compensate for this unreliability, whenever a rendezvous daemon publishes a message to other daemons, it will keep that message for at least 60 seconds. When publishing a message, a daemon attaches a (subject independent) sequence number to that message. A receiving daemon can detect it is missing a message by looking at sequence numbers (recall that messages are delivered to all daemons). When a message has been missed, the publishing daemon is requested to retransmit the message.

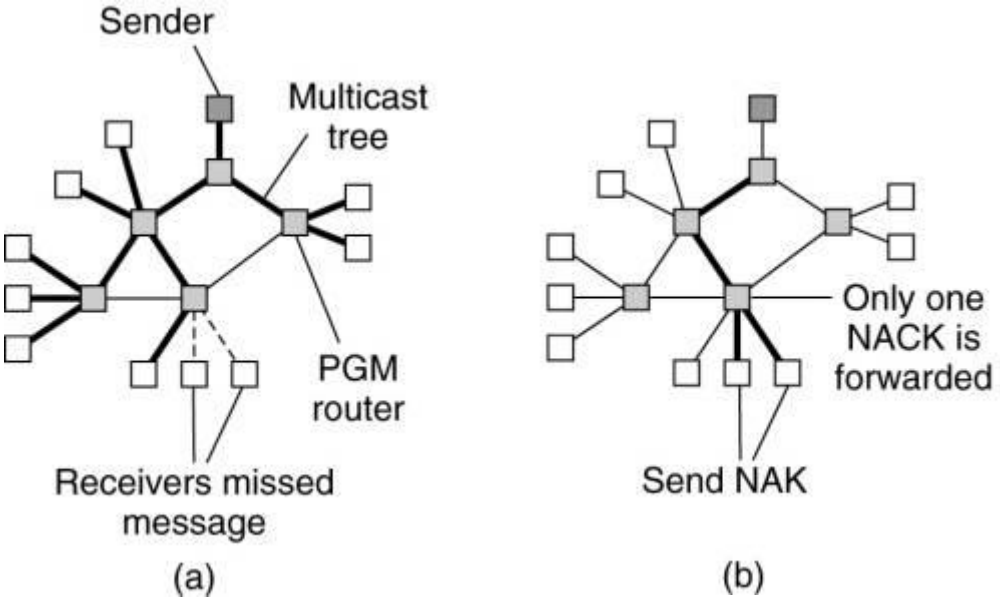
This form of reliable communication cannot prevent that messages may still be lost. For example, if a receiving daemon requests a retransmission of a message that has been published more than 60 seconds ago, the publishing daemon will generally not be able to help recover this lost message. Under normal circumstances, the publishing and subscribing applications will be notified that a communication error has occurred. Error handling is then left to the applications to deal with.

Much of the reliability of communication in TIB/Rendezvous is based on the reliability offered by the underlying network. TIB/Rendezvous also provides reliable multicasting using (unreliable) IP multicasting as its underlying communication means. The scheme followed in TIB/Rendezvous is a transport-level multicast protocol known as Pragmatic General Multicast (PGM), which is described in Speakman et al. (2001). We will discuss PGM briefly.

PGM does not provide hard guarantees that when a message is multicast it will eventually be delivered to each receiver. Fig. 13-16(a) shows a situation in which a message has been multicast along a tree, but it has not been delivered to two receivers. PGM relies on receivers detecting that they have missed messages for which they will send a retransmission request (i.e., a NAK) to the sender. This

request is sent along the reverse path in the multicast tree rooted at the sender, as shown in Fig. 13-16(b). Whenever a retransmission request reaches an intermediate node, that node may possibly have cached the requested message, at which point it will handle the retransmission. Otherwise, the node simply forwards the NAK to the next node toward the sender. The sender is ultimately responsible for retransmitting a message.

Figure 13-16. The principle of PGM. (a) A message is sent along a multicast tree. (b) A router will pass only a single NAK for each message. (c) A message is retransmitted only to receivers that have asked for it.
(This item is displayed on page 615 in the print version)



PGM takes several measures to provide a scalable solution to reliable multicasting. First, if an intermediate node receives several retransmission requests for exactly the same message, only one retransmission request is forwarded toward the sender. In this way, an attempt is made to ensure that only a single NAK reaches the sender, so that a feedback implosion is avoided. We already came across this problem in Chap. 8 when discussing scalability issues in reliable multicasting.
[Page 615]

A second measure taken by PGM is to remember the path through which a NAK traverses from receivers to the sender, as is shown in Fig. 13-16(c). When the sender finally retransmits the requested message, PGM takes care that the message is multicast only to those receivers that had requested retransmission.

Consequently, receivers to which the message had been successfully delivered are not bothered by retransmissions for which they have no use.

Besides the basic reliability scheme and reliable multicasting through PGM, TIB/Rendezvous provides further reliability by means of certified message delivery. In this case, a process uses a special communication channel for sending or receiving messages. The channel has an associated facility, called a ledger, for keeping track of sent and received certified messages. A process that wants to receive certified messages registers itself with the sender of such messages. In effect, registration allows the channel to handle further reliability issues for which the rendezvous daemons provide no support. Most of these issues are hidden from applications and are handled by the channel's implementation.

When a ledger is implemented as a file, it becomes possible to provide reliable message delivery even in the presence of process failures. For example, when a receiving process crashes, all messages it misses until it recovers again are stored in a sender's ledger. Upon recovery, the receiver simply contacts the ledger and requests the missed messages to be retransmitted.

To enable the masking of process failures, TIB/Rendezvous provides a simple means to automatically activate or deactivate processes. In this context, an active process normally responds to all incoming messages, while an inactive one does not. An inactive process is a running process that can handle only special events as we explain shortly.
[Page 616]

Processes can be organized into a group, with each process having a unique rank associated with it. The rank of a process is determined by its (manually assigned) weight, but no two processes in the same group may have the same rank. For each group, TIB/Rendezvous will attempt to have a group-specific number of processes active, called the group's active goal. In many cases, the active goal is set to one so that all communication with a group reduces to a primary-based protocol as discussed in Chap. 7.

An active process regularly sends a message to all other members in the group to announce that it is still up and running. Whenever such a heartbeat message is missing, the middleware will automatically activate the highest-ranked process that is currently inactive. Activation is accomplished by a callback to an action operation that each group member is expected to implement. Likewise, when a previously crashed process recovers again and becomes active, the lowest-ranked currently active process will be automatically deactivated.

To keep consistent with the active processes, special measures need to be taken by an inactive process before it can become active. A simple approach is to let an inactive process subscribe to the same messages as any other group member. An incoming message is processed as usual, but no reactions are ever published. Note that this scheme is akin to active replication.

13.8.2. Fault Tolerance in Shared Dataspaces

When dealing with generative communication, matters become more complicated. As also noted in Tolksdorf and Rowstron (2000), as soon as fault tolerance needs to be incorporated in shared dataspaces, solutions can often become so inefficient that only centralized implementations are feasible. In such cases, traditional solutions are applied, notably using a central server that is backed up in using a simple primary-backup protocol, in combination with checkpointing.

An alternative is to deploy replication more aggressively by placing copies of data items across the various machines. This approach has been adopted in GSpace, essentially deploying the same mechanisms it uses for improving performance through replication. To this end, each node computes its availability, which is then used in computing the availability of a single (replicated) data item (Russello et al., 2006).

To compute its availability, a node regularly writes a timestamp to persistent storage, allowing it to compute the time when it is up, and the time when it was down. More precisely, availability is computed in terms of the mean time to failure (MTTF) and the mean time to repair (MTTR):

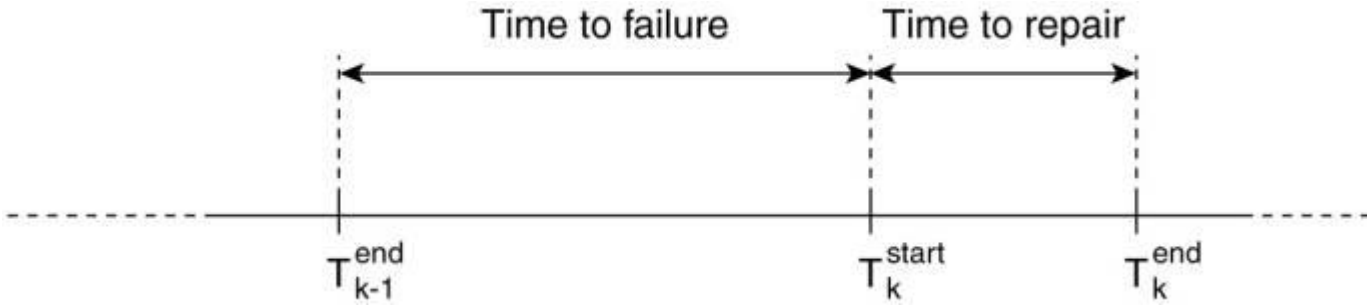
$$Availability\ node = \frac{MTTF}{MTTF+MTTR}$$

[Page 617]

To compute MTTF and MTTR, a node simply looks at the logged timestamps, as shown in Fig. 13-17. This will allow it to compute the averages for the time between failures, leading to an availability of:

$$Availability\ node = \frac{\sum_{k=1}^{n} (T_k^{start} - T_{k-1}^{end})}{\sum_{k=1}^{n} (T_k^{start} - T_{k-1}^{end}) + \sum_{k=1}^{n} (T_k^{end} - T_k^{start})}$$

Figure 13-17. The time line of a node experiencing failures.



Note that it is necessary to regularly log timestamps and that T_k^{start} can be taken only as a best estimate of when a crash occurred. However, the thus computed availability will be pessimistic, as the actual time that a node crashed for the kth time will be slightly later than T_k^{start}. Also, instead of taking averages since the beginning, it is also possible to take only the last N crashes into account.

In GSpace, each type of data item has an associated primary node that is responsible for computing that type's availability. Given that a data item is replicated across m nodes, its availability is computed by considering the availability ai of each of the m nodes leading to:

$$Availability\ data\ item = 1 - \prod_{k=1}^{m} (1 - a_i)$$

By simply taking the availability of a data item into account, as well as those of all nodes, the primary can compute an optimal placement for a data item that will satisfy the availability requirements for a data item. In addition, it can also take other factors into account, such as bandwidth usage and CPU loads. Note that placement may change over time if these factors fluctuate.

13.9. Security

Security in coordination-based systems poses a difficult problem. On the one hand we have stated that processes should be referentially decoupled, but on the other hand we should also ensure the integrity and confidentiality of data. This

security is normally implemented through secure (multicast) channels, which effectively require that senders and receivers can authenticate each other. Such authentication violates referential decoupling.
[Page 618]

To solve this problem there are different approaches. One common approach is to set up a network of brokers that handle the processing of data and subscriptions. Client processes will then contact the brokers, who then take care of authentication and authorization. Note that such an approach does require that the clients trust the brokers. However, as we shall see later, by differentiating between types of brokers, it is not necessary that a client has to trust all brokers comprising the system.

By nature of data coordination, authorization naturally translates to confidentiality issues. We will now take a closer look at these issues, following the discussion as presented in Wang et al. (2002).

13.9.1. Confidentiality

One important difference between many distributed systems and coordination-based ones is that in order to provide efficiency, the middleware needs to inspect the content of published data. Without being able to do so, the middleware can essentially only flood data to all potential subscribers. This poses the problem of information confidentiality which refers to the fact that it is sometimes important to disallow the middleware to inspect published data. This problem can be circumvented through end-to-end encryption; the routing substrate only sees source and destination addresses.

If published data items are structured in the sense that every item contains multiple fields, it is possible to deploy partial secrecy. For example, data regarding real estate may need to be shipped between agents of the same office with branches at different locations, but without revealing the exact address of the property. To allow for content-based routing, the address field could be encrypted, while the description of the property could be published in the clear. To this end, Khurana and Koleva (2006) propose to use a per-field encryption scheme as introduced in Bertino and Ferrari (2002). In this case, the agents belonging to the same branch would share the secret key for decrypting the address field. Of course, this violates referential decoupling, but we will discuss a potential solution to this problem later.

More problematic is the case when none of the fields may be disclosed to the middleware in plaintext. The only solution that remains is that content-based routing takes place on the encrypted data. As routers get to see only encrypted

data, possibly on a per-field basis, subscriptions will need to be encoded in such a way that partial matching can take place. Note that a partial match is the basis that a router uses to decide which outgoing link a published data item should be forwarded on.

This problem comes very close to querying and searching through encrypted data, something clearly next to impossible to achieve. As it turns out, maintaining a high degree of secrecy while still offering reasonable performance is known to be very difficult (Kantarcioglu and Clifton, 2005). One of the problems is that if per-field encryption is used, it becomes much easier to find out what the data is all about.
[Page 619]

Having to work on encrypted data also brings up the issue of subscription confidentiality, which refers to the fact that subscriptions may not be disclosed to the middleware either. In the case of subject-based addressing schemes, one solution is to simply use per-field encryption and apply matching on a strict field-by-field basis. Partial matching can be accommodated in the case of compound keywords, which can be represented as encrypted sets of their constituents. A subscriber would then send encrypted forms of such constituents and let the routers check for set membership, as also suggested by Raiciu and Rosenblum (2005). As it turns out, it is even possible to support range queries, provided an efficient scheme can be devised for representing intervals. A potential solution is discussed in Li et al. (2004a).

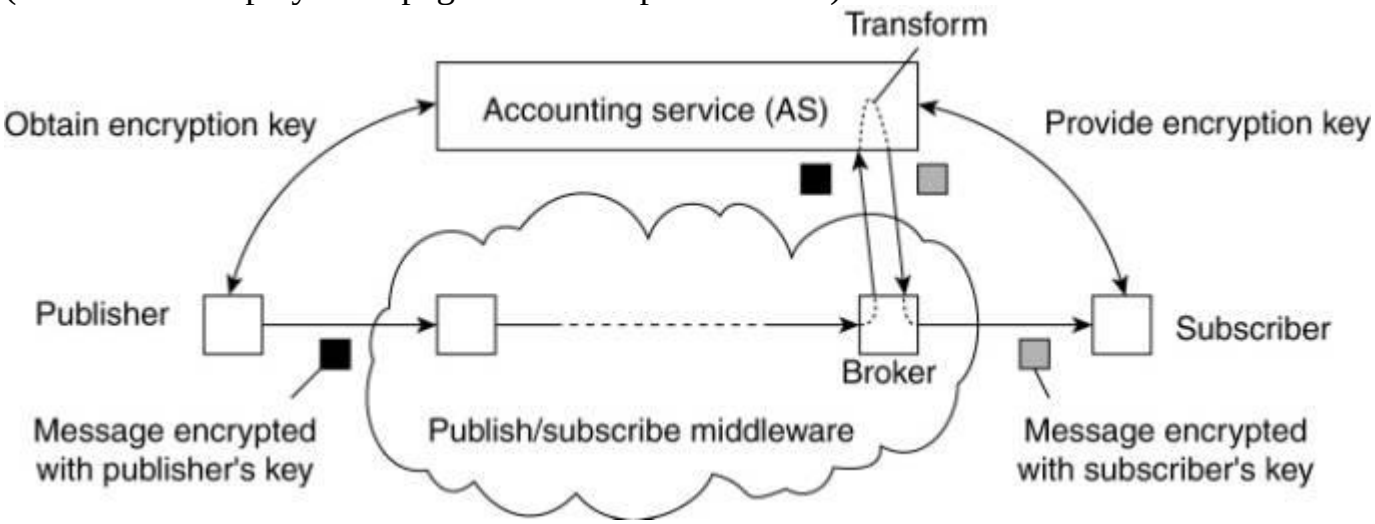
Finally, publication confidentiality is also an issue. In this case, we are touching upon the more traditional access control mechanisms in which certain processes should not even be allowed to see certain messages. In such cases, publishers may want to explicitly restrict the group of possible subscribers. In many cases, this control can be exerted out-of-band at the level of the publishing and subscribing applications. However, it may convenient that the middleware offers a service to handle such access control.

Decoupling Publishers from Subscribers

If it is necessary to protect data and subscriptions from the middleware, Khurana and Koleva (2006) propose to make use of a special accounting service (AS), which essentially sits between clients (publishers and subscribers) and the actual publish/subscribe middleware. The basic idea is to decouple publishers from subscribers while still providing information confidentiality. In their scheme, subscribers register their interest in specific data items, which are subsequently routed as usual. The data items are assumed to contain fields that have been encrypted. To allow for decryption, once a message should be delivered to a

subscriber, the router passes it to the accounting service where it is transformed into a message that only the subscriber can decrypt. This scheme is shown in Fig. 13-18.

Figure 13-18. Decoupling publishers from subscribers using an additional trusted service.
(This item is displayed on page 620 in the print version)



A publisher registers itself at any node of the publish/subscribe network, that is, at a broker. The broker forwards the registration information to the accounting service which then generates a public key to be used by the publisher, and which is signed by the AS. Of course, the AS keeps the associated private key to itself. When a subscriber registers, it provides an encryption key that is forwarded by the broker. It is necessary to go through a separate authentication phase to ensure that only legitimate subscribers register. For example, brokers should generally not be allowed to subscribe for published data.
[Page 620]

Ignoring many details, when a data item is published, its critical fields will have been encrypted by the publisher. When the data item arrives at a broker who wishes to pass it on to a subscriber, the former requests the AS to transform the message by first decrypting it, and then encrypt it with the key provided by the subscriber. In this way, the brokers will never get to know about content that should be kept secret, while at the same time, publishers and subscribers need not share key information.

Of course, it is crucial that accounting service itself can scale. Various measures can be taken, but one reasonable approach is to introduce realms in a similar way

that Kerberos does. In this case, messages in transmission may need to be transformed by re-encrypting them using the public key of a foreign accounting service. For details, we refer the interested reader to (Khurana and Koleva, 2006).

13.9.2. Secure Shared Dataspaces

Very little work has been done when it comes to making shared dataspaces secure. A common approach is to simply encrypt the fields of data items and let matching take place only when decryption succeeds and content matches with a subscription. This approach is described in Vitek et al. (2003). One of the major problems with this approach is that keys may need to be shared between publishers and subscribers, or that the decryption keys of the publishers should be known to authorized subscribers.

Of course, if the shared dataspace is trusted (i.e., the processes implementing the dataspace are allowed to see the content of tuples), matters become much simpler. Considering that most implementations make use of only a single server, extending that server with authentication and authorization mechanisms is often the approach followed in practice.

Chapter 13. Distributed Coordination-Based Systems

In the previous chapters we took a look at different approaches to distributed systems, in each chapter focusing on a single data type as the basis for distribution. The data type, being either an object, file, or (Web) document, has its origins in nondistributed systems. It is adapted for distributed systems in such a way that many issues about distribution can be made transparent to users and developers.

In this chapter we consider a generation of distributed systems that assume that the various components of a system are inherently distributed and that the real problem in developing such systems lies in coordinating the activities of different components. In other words, instead of concentrating on the transparent distribution of components, emphasis lies on the coordination of activities between those components.

We will see that some aspects of coordination have already been touched upon in the previous chapters, especially when considering event-based systems. As it turns out, many conventional distributed systems are gradually incorporating mechanisms that play a key role in coordination-based systems.

Before taking a look at practical examples of systems, we give a brief introduction to the notion of coordination in distributed systems.

13.1. Introduction to Coordination Models

Key to the approach followed in coordination-based systems is the clean separation between computation and coordination. If we view a distributed system as a collection of (possibly multithreaded) processes, then the computing part of a distributed system is formed by the processes, each concerned with a specific computational activity, which in principle, is carried out independently from the activities of other processes.
[Page 590]

In this model, the coordination part of a distributed system handles the communication and cooperation between processes. It forms the glue that binds the activities performed by processes into a whole (Gelernter and Carriero, 1992). In distributed coordination-based systems, the focus is on how coordination between the processes takes place.

Cabri et al. (2000) provide a taxonomy of coordination models for mobile agents that can be applied equally to many other types of distributed systems. Adapting their terminology to distributed systems in general, we make a distinction between models along two different dimensions, temporal and referential, as shown in Fig. 13-1.

Figure 13-1. A taxonomy of coordination models (adapted from Cabri et al., 2000).

Referential \ Temporal	Coupled	Decoupled
Coupled	Direct	Mailbox
Decoupled	Meeting oriented	Generative communication

When processes are temporally and referentially coupled, coordination takes place in a direct way, referred to as direct coordination. The referential coupling generally appears in the form of explicit referencing in communication. For example, a process can communicate only if it knows the name or identifier of the other processes it wants to exchange information with. Temporal coupling means that processes that are communicating will both have to be up and running.

This coupling is analogous to the transient message-oriented communication we discussed in Chap. 4.

A different type of coordination occurs when processes are temporally decoupled, but referentially coupled, which we refer to as mailbox coordination. In this case, there is no need for two communicating processes to execute at the same time in order to let communication take place. Instead, communication takes place by putting messages in a (possibly shared) mailbox. This situation is analogous to persistent message-oriented communication as described in Chap. 4. It is necessary to explicitly address the mailbox that will hold the messages that are to be exchanged. Consequently, there is a referential coupling.

The combination of referentially decoupled and temporally coupled systems form the group of models for meeting-oriented coordination. In referentially decoupled systems, processes do not know each other explicitly. In other words, when a process wants to coordinate its activities with other processes, it cannot directly refer to another process. Instead, there is a concept of a meeting in which processes temporarily group together to coordinate their activities. The model prescribes that the meeting processes are executing at the same time.
[Page 591]

Meeting-based systems are often implemented by means of events, like the ones supported by object-based distributed systems. In this chapter, we discuss another mechanism for implementing meetings, namely publish/subscribe systems. In these systems, processes can subscribe to messages containing information on specific subjects, while other processes produce (i.e., publish) such messages. Most publish/subscribe systems require that communicating processes are active at the same time; hence there is a temporal coupling. However, the communicating processes may otherwise remain anonymous.

The most widely-known coordination model is the combination of referentially and temporally decoupled processes, exemplified by generative communication as introduced in the Linda programming system by Gelernter (1985). The key idea in generative communication is that a collection of independent processes make use of a shared persistent dataspace of tuples. Tuples are tagged data records consisting of a number (but possibly zero) typed fields. Processes can put any type of record into the shared dataspace (i.e., they generate communication records). Unlike the case with blackboards, there is no need to agree in advance on the structure of tuples. Only the tag is used to distinguish between tuples representing different kinds of information.

An interesting feature of these shared dataspaces is that they implement an associative search mechanism for tuples. In other words, when a process wants

to extract a tuple from the dataspace, it essentially specifies (some of) the values of the fields it is interested in. Any tuple that matches that specification is then removed from the dataspace and passed to the process. If no match could be found, the process can choose to block until there is a matching tuple. We defer the details on this coordination model to later when discussing concrete systems.

We note that generative communication and shared dataspaces are often also considered to be forms of publish/subscribe systems. In what follows, we shall adopt this commonality as well. A good overview of publish/subscribe systems (and taking a rather broad perspective) can be found in Eugster et al. (2003). In this chapter we take the approach that in these systems there is at least referential decoupling between processes, but preferably also temporal decoupling.

13.2. Architectures

An important aspect of coordination-based systems is that communication takes place by describing the characteristics of data items that are to be exchanged. As a consequence, naming plays a crucial role. We return to naming later in this chapter, but for now the important issue is that in many cases, data items are not explicitly identified by senders and receivers.
[Page 592]

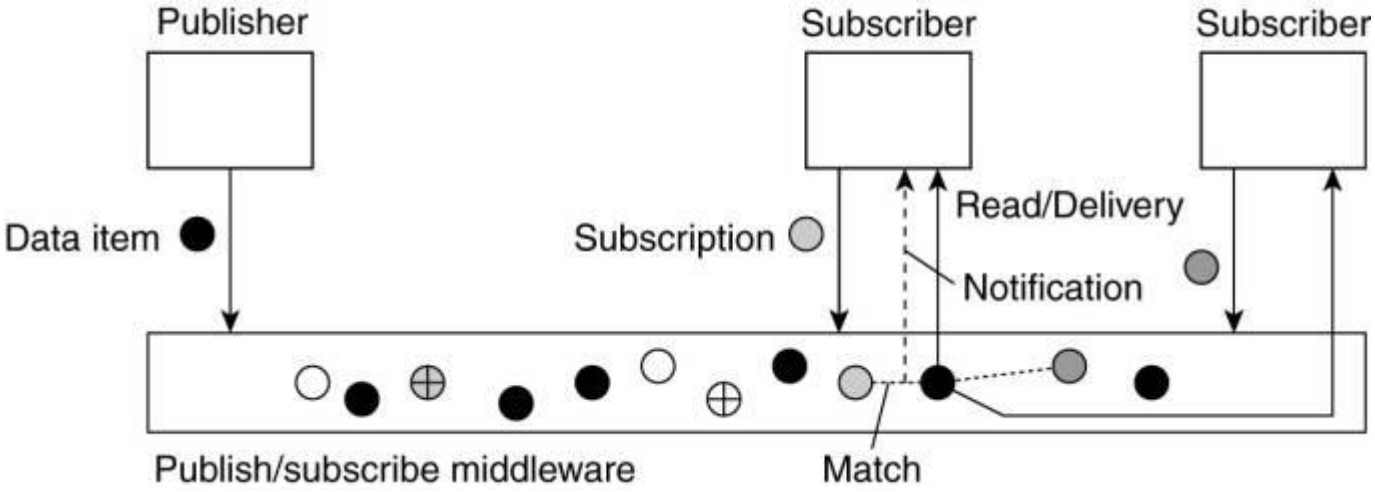
13.2.1. Overall Approach

Let us first assume that data items are described by a series of attributes. A data item is said to be published when it is made available for other processes to read. To that end, a subscription needs to be passed to the middleware, containing a description of the data items that the subscriber is interested in. Such a description typically consists of some (attribute, value) pairs, possibly combined with (attribute, range) pairs. In the latter case, the specified attribute is expected to take on values within a specified range. Descriptions can sometimes be given using all kinds of predicates formulated over the attributes, very similar in nature to SQL-like queries in the case of relational databases. We will come across these types of descriptors later in this chapter.

We are now confronted with a situation in which subscriptions need to be matched against data items, as shown in Fig. 13-2. When matching succeeds, there are two possible scenarios. In the first case, the middleware may decide to forward the published data to its current set of subscribers, that is, processes with a matching subscription. As an alternative, the middleware can also forward a

notification at which point subscribers can execute a read operation to retrieve the published data item.

Figure 13-2. The principle of exchanging data items between publishers and subscribers.



In those cases in which data items are immediately forwarded to subscribers, the middleware will generally not offer storage of data. Storage is either explicitly handled by a separate service, or is the responsibility of subscribers. In other words, we have a referentially decoupled, but temporally coupled system.

This situation is different when notifications are sent so that subscribers need to explicitly read the published data. Necessarily, the middleware will have to store data items. In these situations there are additional operations for data management. It is also possible to attach a lease to a data item such that when the lease expires that the data item is automatically deleted.

In the model described so far, we have assumed that there is a fixed set of n attributes a1, . . . , an that is used to describe data items. In particular, each published data item is assumed to have an associated vector <(a1,v1),...,(an,vn)> of (attribute, value) pairs. In many coordination-based systems, this assumption is false. Instead, what happens is that events are published, which can be viewed as data items with only a single specified attribute.
[Page 593]

Events complicate the processing of subscriptions. To illustrate, consider a subscription such as "notify when room R4.20 is unoccupied and the door is unlocked." Typically, a distributed system supporting such subscriptions can be implemented by placing independent sensors for monitoring room occupancy

(e.g., motion sensors) and those for registering the status of a door lock. Following the approach sketched so far, we would need to compose such primitive events into a publishable data item to which processes can then subscribe. Event composition turns out to be a difficult task, notably when the primitive events are generated from sources dispersed across the distributed system.

Clearly, in coordination-based systems such as these, the crucial issue is the efficient and scalable implementation of matching subscriptions to data items, along with the construction of relevant data items. From the outside, a coordination approach provides lots of potential for building very large-scale distributed systems due to the strong decoupling of processes. On the other hand, as we shall see next, devising scalable implementations without losing this independence is not a trivial exercise.

13.2.2. Traditional Architectures

The simplest solution for matching data items against subscriptions is to have a centralized client-server architecture. This is a typical solution currently adopted by many publish/subscribe systems, including IBM's WebSphere (IBM, 2005c) and popular implementations for Sun's JMS (Sun Microsystems, 2004a). Likewise, implementations for the more elaborate generative communication models such as Jini (Sun Microsystems, 2005b) and JavaSpaces (Freeman et al., 1999) are mostly based on central servers. Let us take a look at two typical examples.

Example: Jini and JavaSpaces

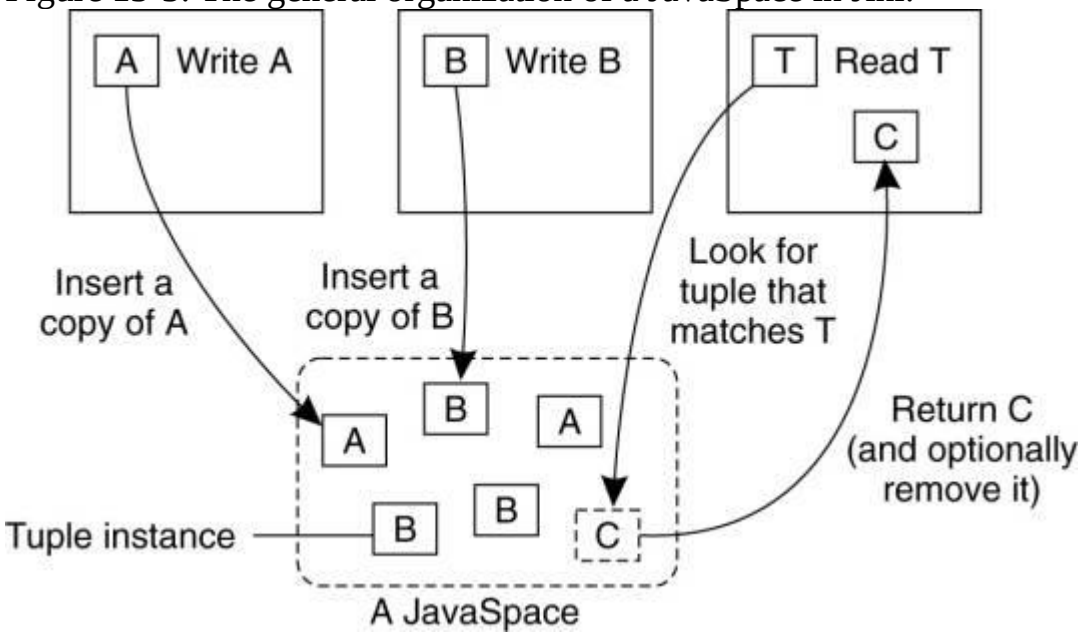
Jini is a distributed system that consists of a mixture of different but related elements. It is strongly related to the Java programming language, although many of its principles can be implemented equally well in other languages. An important part of the system is formed by a coordination model for generative communication. Jini provides temporal and referential decoupling of processes through a coordination system called JavaSpaces (Freeman et al., 1999), derived from Linda. A JavaSpace is a shared dataspace that stores tuples representing a typed set of references to Java objects. Multiple JavaSpaces may coexist in a single Jini system.

Tuples are stored in serialized form. In other words, whenever a process wants to store a tuple, that tuple is first marshaled, implying that all its fields are marshaled as well. As a consequence, when a tuple contains two different fields that refer to the same object, the tuple as stored in a JavaSpace implementation will hold two marshaled copies of that object.

[Page 594]

A tuple is put into a JavaSpace by means of a write operation, which first marshals the tuple before storing it. Each time the write operation is called on a tuple, another marshaled copy of that tuple is stored in the JavaSpace, as shown in Fig. 13-3. We will refer to each marshaled copy as a tuple instance.

Figure 13-3. The general organization of a JavaSpace in Jini.



The interesting aspect of generative communication in Jini is the way that tuple instances are read from a JavaSpace. To read a tuple instance, a process provides another tuple that it uses as a template for matching tuple instances as stored in a JavaSpace. Like any other tuple, a template tuple is a typed set of object references. Only tuple instances of the same type as the template can be read from a JavaSpace. A field in the template tuple either contains a reference to an actual object or contains the value NULL. For example, consider the class

```
class public Tuple implements Entry {
  public Integer id, value;
    public Tuple(Integer id, Integer value){this.id = id; this.value =
value}
}
```

Then a template declared as

Tuple template = new Tuple(null, new Integer(42))

will match the tuple

Tuple item = new Tuple("MyName", new Integer(42))

To match a tuple instance in a JavaSpace against a template tuple, the latter is marshaled as usual, including its NULL fields. For each tuple instance of the same type as the template, a field-by-field comparison is made with the template tuple. Two fields match if they both have a copy of the same reference or if the field in the template tuple is NULL. A tuple instance matches a template tuple if there is a pairwise matching of their respective fields.
[Page 595]

When a tuple instance is found that matches the template tuple provided as part of a read operation, that tuple instance is unmarshaled and returned to the reading process. There is also a take operation that additionally removes the tuple instance from the JavaSpace. Both operations block the caller until a matching tuple is found. It is possible to specify a maximum blocking time. In addition, there are variants that simply return immediately if no matching tuple existed.

Processes that make use of JavaSpaces need not coexist at the same time. In fact, if a JavaSpace is implemented using persistent storage, a complete Jini system can be brought down and later restarted without losing any tuples.

Although Jini does not support it, it should be clear that having a central server allows subscriptions to be fairly elaborate. For example, at the moment two nonnull fields match if they are identical. However, realizing that each field represents an object, matching could also be evaluated by executing an object-specific comparison operator [see also Picco et al. (2005)]. In fact, if such an operator can be overridden by an application, more-or-less arbitrary comparison semantics can be implemented. It is important to note that such comparisons may require an extensive search through currently stored data items. Such searches cannot be easily efficiently implemented in a distributed way. It is exactly for this reason that when elaborate matching rules are supported we will generally see only centralized implementations.

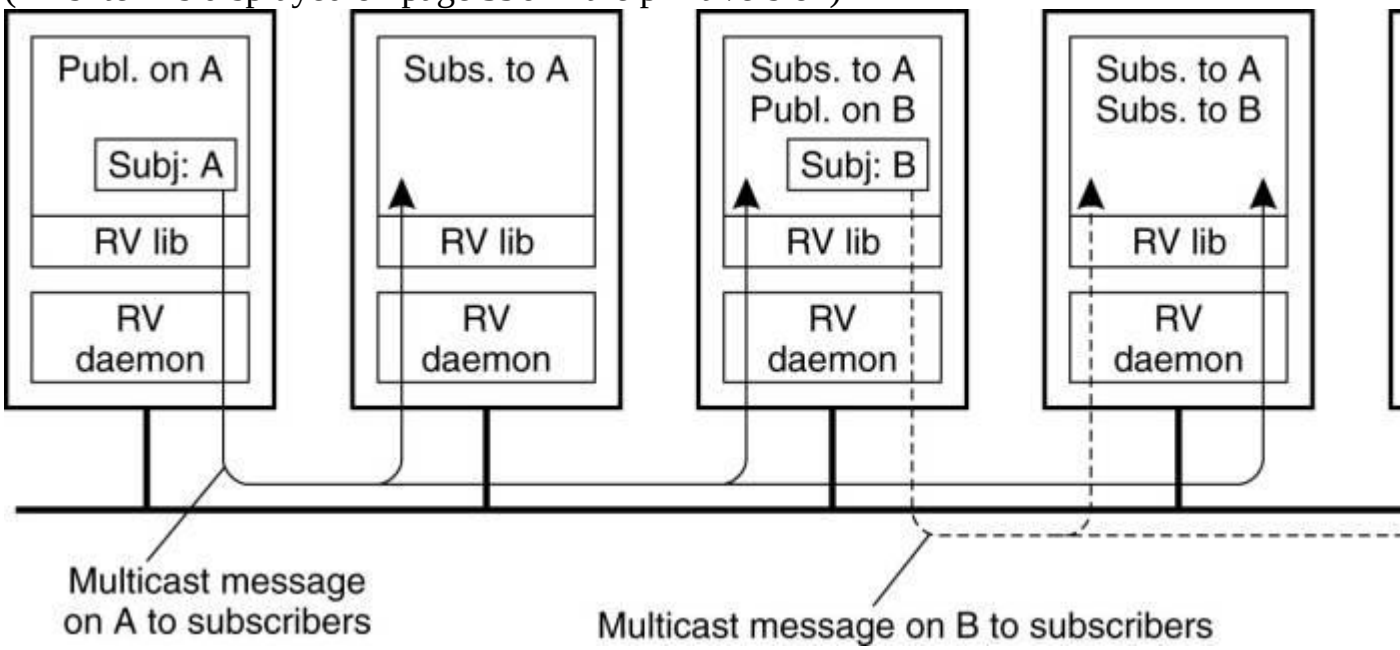
Another advantage of having a centralized implementation is that it becomes easier to implement synchronization primitives. For example, the fact that a process can block until a suitable data item is published, and then subsequently

execute a destructive read by which the matching tuple is removed, offers facilities for process synchronization without processes needing to know each other. Again, synchronization in decentralized systems is inherently difficult as we also discussed in Chap. 6. We will return to synchronization below.

Example: TIB/Rendezvous

An alternative solution to using central servers is to immediately disseminate published data items to the appropriate subscribers using multicasting. This principle is used in TIB/Rendezvous, of which the basic architecture is shown in Fig. 13-4 (TIBCO, 2005) In this approach, a data item is a message tagged with a compound keyword describing its content, such as news.comp.os.books. A subscriber provides (parts of) a keyword, or indicating the messages it wants to receive, such as news.comp.*.books. These keywords are said to indicate the subject of a message.

Figure 13-4. The principle of a publish/subscribe system as implemented in TIB/Rendezvous.
(This item is displayed on page 596 in the print version)



Fundamental to its implementation is the use of broadcasting common in local-area networks, although it also uses more efficient communication facilities when possible. For example, if it is known exactly where a subscriber resides, point-to-point messages will generally be used. Each host on such a network will run a rendezvous daemon, which takes care that messages are sent and delivered according to their subject. Whenever a message is published, it is multicast to

each host on the network running a rendezvous daemon. Typically, multicasting is implemented using the facilities offered by the underlying network, such as IP-multicasting or hardware broadcasting.
[Page 596]

Processes that subscribe to a subject pass their subscription to their local daemon. The daemon constructs a table of (process, subject), entries and whenever a message on subject S arrives, the daemon simply checks in its table for local subscribers, and forwards the message to each one. If there are no subscribers for S, the message is discarded immediately.

When using multicasting as is done in TIB/Rendezvous, there is no reason why subscriptions cannot be elaborate and be more than string comparison as is currently the case. The crucial observation here is that because messages are forwarded to every node anyway, the potentially complex matching of published data against subscriptions can be done entirely locally without further network communication. However, as we shall discuss later, simple comparison rules are required whenever matching across wide-area networks is needed.

13.2.3. Peer-to-Peer Architectures

The traditional architectures followed by most coordination-based systems suffer from scalability problems (although their commercial vendors will state otherwise). Obviously, having a central server for matching subscriptions to published data cannot scale beyond a few hundred clients. Likewise, using multicasting requires special measures to extend beyond the realm of local-area networks. Moreover, if scalability is to be guaranteed, further restrictions on describing subscriptions and data items may be necessary.
[Page 597]

Much research has been spent on realizing coordination-based systems using peer-to-peer technology. Straightforward implementations exist for those cases in which keywords are used, as these can be hashed to unique identifiers for published data. This approach has also been used for mapping (attribute, value) pairs to identifiers. In these cases, matching reduces to a straightforward lookup of an identifier, which can be efficiently implemented in a DHT-based system. This approach works well for the more conventional publish/subscribe systems as illustrated by Tam and Jacobsen (2003), but also for generative communication (Busi et al., 2004).

Matters become complicated for more elaborate matching schemes. Notoriously difficult are the cases in which ranges need to be supported and only very few

proposals exist. In the following, we discuss one such proposal, devised by one of the authors and his colleagues (Voulgaris et al., 2006).

Example: A Gossip-Based Publish/Subscribe System

Consider a publish/subscribe system in which data items can be described by means of N attributes a1, . . . , aN whose value can be directly mapped to a floating-point number. Such values include, for example, floats, integers, enumerations, booleans, and strings. A subscription s takes the form of a tuple of (attribute, value/range) pairs, such as

$s = \langle a1 \longrightarrow 3.0, a4 \longrightarrow [0.0, 0.5) \rangle$

In this example, s specifies that a1 should be equal to 3.0, and a4 should lie in the interval [0.0, 0.5). Other attributes are allowed to take on any value. For clarity, assume that every node i enters only one subscription si.

Note that each subscription si actually specifies a subset Si in the N-dimensional space of floating-point numbers. Such a subset is also called a hyperspace. For the system as a whole, only published data whose description falls in the union $S = \bigcup Si$ of these hyperspaces is of interest. The whole idea is to automatically partition S into M disjoint hyperspaces S1, . . . , SM such that each falls completely in one of the subscription hyperspaces Si, and together they cover all subscriptions. More formally, we have that:

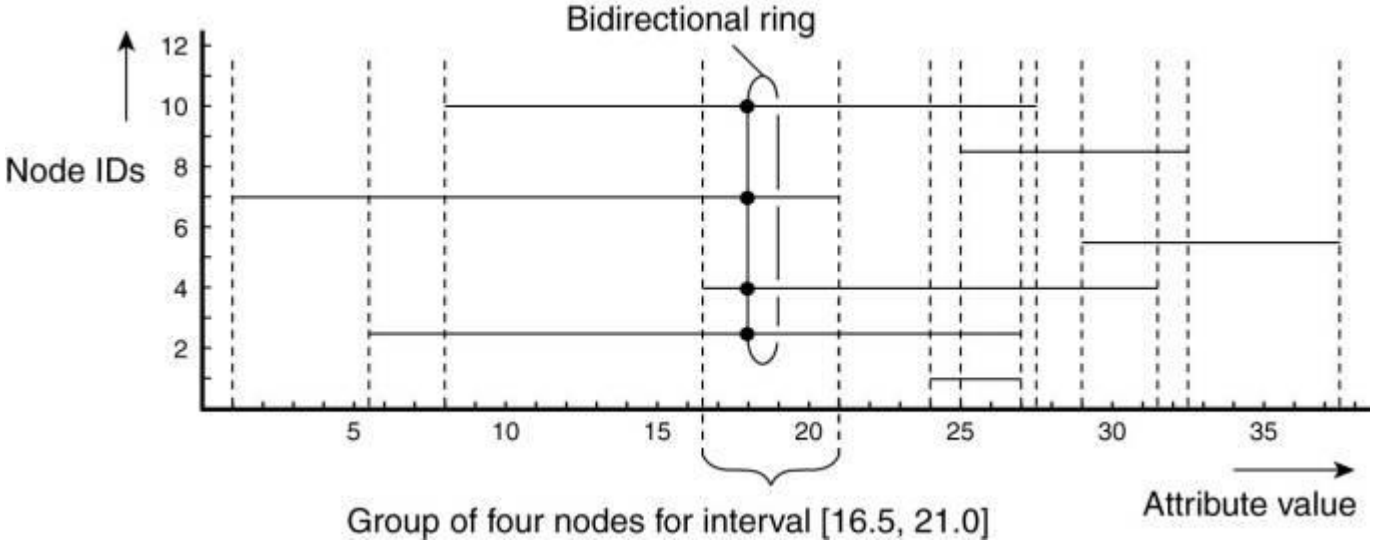
$$(Sm \cap Si \neq \emptyset) \Longrightarrow (Sm \subseteq Si)$$

Moreover, the system keeps M minimal in the sense that there is no partitioning with fewer parts Sm. The whole idea is to register, for each hyperspace Sm, exactly those nodes i for which $Sm \subseteq Si$. In that case, when a data item is published, the system need merely find the Sm to which that item belongs, from which point it can forward the item to the associated nodes.
[Page 598]

To this end, nodes regularly exchange subscriptions using an epidemic protocol. If two nodes i and j notice that their respective subscriptions intersect, that is $Sij \equiv Si \cap Sj \neq \emptyset$ they will record this fact and keep references to each other. If they discover a third node k with $Sijk \equiv Sij \cap Sk \neq \emptyset$, the three of them will connect to each other so that a data item d from Sijk can be efficiently disseminated. Note that if $Sij - Sijk \neq \emptyset$, nodes i and j will maintain their mutual references, but now associate it strictly with Sij - Sijk.

In essence, what we are seeking is a means to cluster nodes into M different groups, such that nodes i and j belong to the same group if and only if their subscriptions Si and Sj intersect. Moreover, nodes in the same group should be organized into an overlay network that will allow efficient dissemination of a data item in the hyperspace associated with that group. This situation for a single attribute is sketched in Fig. 13-5.

Figure 13-5. Grouping nodes for supporting range queries in a peer-to-peer publish/subscribe system.



Here, we see a total of seven nodes in which the horizontal line for node i indicates its range of interest for the value of the single attribute. Also shown is the grouping of nodes into disjoint ranges of interests for values of the attribute. For example, nodes 3, 4, 7, and 10 will be grouped together representing the interval [16.5, 21.0]. Any data item with a value in this range should be disseminated to only these four nodes.

To construct these groups, the nodes are organized into a gossip-based unstructured network. Each node maintains a list of references to other neighbors (i.e., a partial view), which it periodically exchanges with one of its neighbors as described in Chap. 2. Such an exchange will allow a node to learn about random other nodes in the system. Every node keeps track of the nodes it discovers with overlapping interests (i.e., with an intersecting subscription).

At a certain moment, every node i will generally have references to other nodes with overlapping interests. As part of exchanging information with a node j, node i orders these nodes by their identifiers and selects the one with the lowest

identifier i1 > j, such that its subscription overlaps with that of node j, that is, $S_{j,i1} \equiv S_{i1} \cap S_j \neq \emptyset$.
[Page 599]

The next one to be selected is i2 > i1 such that its subscription also overlaps with that of j, but only if it contains elements not yet covered by node i1. In other words, we should have that $S_{j,i1,i2} \equiv (S_{i2} - S_{j,i1}) \cap S_j \neq \emptyset$. This process is repeated until all nodes that have an overlapping interest with node i have been inspected, leading to an ordered list $i1 < i2 < \ldots < in$. Note that a node ik is in this list because it covers a region R of common interest to node i and j not yet jointly covered by nodes with a lower identifier than ik. In effect, node ik is the first node that node j should forward a data item to that falls in this unique region R. This procedure can be expanded to let node i construct a bidirectional ring. Such a ring is also shown in Fig. 13-5.

Whenever a data item d is published, it is disseminated as quickly as possible to any node that is interested in it. As it turns out, with the information available at every node finding a node i interested in d is simple. From there on, node i need simply forward d along the ring of subscribers for the particular range that d falls into. To speed up dissemination, short-cuts are maintained for each ring as well. Details can be found in Voulgaris et al. (2006).

Discussion

An approach somewhat similar to this gossip-based solution in the sense that it attempts to find a partitioning of the space covered by the attribute's values, but which uses a DHT-based system is described in Gupta et al. (2004). In another proposal described in Bharambe (2004), each attribute ai is handled by a separate process Pi, which in turn partitions the range of its attribute across multiple processes. When a data item d is published, it is forwarded to each Pi, where it is subsequently stored at the process responsible for the d's value of ai.

All these approaches are illustrative for the complexity when mapping a nontrivial publish/subscribe system to a peer-to-peer network. In essence, this complexity comes from the fact that supporting search in attribute-based naming systems is inherently difficult to establish in a decentralized fashion. We will again come across these difficulties when discussing replication.

13.2.4. Mobility and Coordination

A topic that has received considerable attention in the literature is how to combine publish/subscribe solutions with node mobility. In many cases, it is assumed that there is a fixed basic infrastructure with access points for mobile nodes. Under

these assumptions, the issue becomes how to ensure that published messages are not delivered more than once to a subscriber who switches access points. One practical solution to this problem is to let subscribers keep track of the messages they have already received and simply discard duplicates. Alternative, but more intricate solutions comprise routers that keep track of which messages have been sent to which subscribers (see, e.g., Caporuscio et al., 2003).
[Page 600]

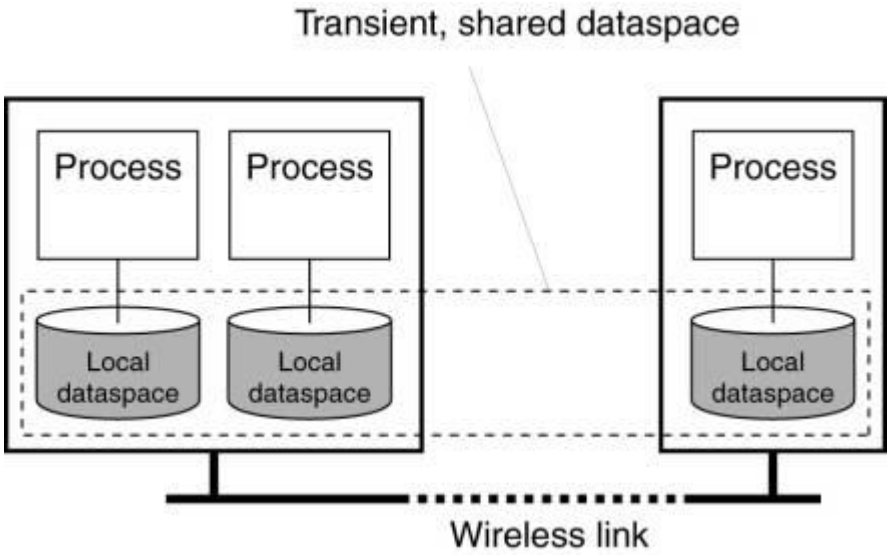
Example: Lime

In the case of generative communication, several solutions have been proposed to operate a shared dataspace in which (some of) the nodes are mobile. A canonical example in this case is Lime (Murphy et al., 2001), which strongly resembles the JavaSpace model we discussed previously.

In Lime, each process has its own associated dataspace, but when processes are in each other's proximity such that they are connected, their dataspaces become shared. Theoretically, being connected can mean that there is a route in a joint underlying network that allows two processes to exchange data. In practice, however, it either means that two processes are temporarily located on the same physical host, or their respective hosts can communicate with each other through a (single hop) wireless link. Formally, the processes should be member of the same group and use the same group communication protocol.

The local dataspaces of connected processes form a transiently shared dataspace that will allow processes to exchange tuples, as shown in Fig. 13-6. For example, when a process P executes a write operation, the associated tuple is stored in the process's local dataspace. In principle, it stays there until there is a matching take operation, possibly from another process that is now in the same group as P. In this way, the fact that we are actually dealing with a completely distributed shared dataspace is transparent for participating processes. However, Lime also allows breaking this transparency by specifying exactly for whom a tuple is intended. Likewise, read and take operations can have an additional parameter specifying from which process a tuple is expected.

Figure 13-6. Transient sharing of local dataspaces in Lime.



To better control how tuples are distributed, dataspaces can carry out what are known as reactions. A reaction specifies an action to be executed when a tuple matching a given template is found in the local dataspace. Each time a dataspace changes, an executable reaction is selected at random, often leading to a further modification of the dataspace. Reactions span the current shared dataspace, but there are several restrictions to ensure that they can be executed efficiently. For example, in the case of weak reactions, it is only guaranteed that the associated actions are eventually executed, provided the matching data is still accessible.
[Page 601]

The idea of reactions has been taken a step further in TOTA, where each tuple has an associated code fragment telling exactly how that tuple should be moved between dataspaces, possibly also including transformations (Mamei and Zambonelli, 2004).

13.3. Processes

There is nothing really special about the processes used in publish/subscribe systems. In most cases, efficient mechanisms need to be deployed for searching in a potentially large collection of data. The main problem is devising schemes that work well in distributed environments. We return to this issue below when discussing consistency and replication.

13.4. Communication

Communication in many publish/subscribe systems is relatively simple. For example, in virtually every Java-based system, all communication proceeds through remote method invocations. One important problem that needs to be handled when publish/subscribe systems are spread across a wide-area system is that published data should reach only the relevant subscribers. As we described above, using a self-organizing method by which nodes in a peer-to-peer system are automatically clustered, after which dissemination takes place per cluster is one solution. An alternative solution is to deploy content-based routing.

13.4.1. Content-Based Routing

In content-based routing, the system is assumed to be built on top of a point-to-point network in which messages are explicitly routed between nodes. Crucial in this setup is that routers can take routing decisions by considering the content of a message. More precisely, it is assumed that each message carries a description of its content, and that this description can be used to cut-off routes for which it is known that they do not lead to receivers interested in that message.

A practical approach toward content-based routing is proposed in Carzaniga et al. (2004). Consider a publish/subscribe system consisting of N servers to which clients (i.e., applications) can send messages, or from which they can read incoming messages. We assume that in order to read messages, an application will have previously provided the server with a description of the kind of data it is interested in. The server, in turn, will notify the application when relevant data has arrived.

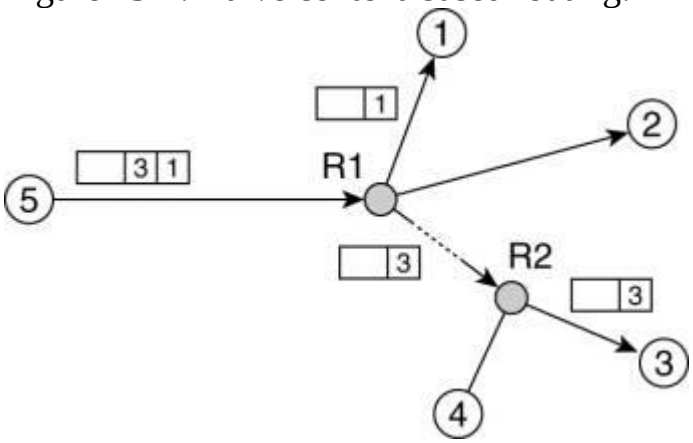
Carzaniga et al. propose a two-layered routing scheme in which the lowest layer consists of a shared broadcast tree connecting the N servers. There are various ways for setting up such a tree, ranging from network-level multicast support to application-level multicast trees as we discussed in Chap. 4. Here, we also assume that such a tree has been set up with the N servers as end nodes, along with a collection of intermediate nodes forming the routers. Note that the distinction between a server and a router is only a logical one: a single machine may host both kinds of processes.

Consider first two extremes for content-based routing, assuming we need to support only simple subject-based publish/subscribe in which each message is tagged with a unique (noncompound) keyword. One extreme solution is to send each published message to every server, and subsequently let the server check

whether any of its clients had subscribed to the subject of that message. In essence, this is the approach followed in TIB/Rendezvous.

The other extreme solution is to let every server broadcast its subscriptions to all other servers. As a result, every server will be able to compile a list of (subject, destination) pairs. Then, whenever an application submits a message on subject s, its associated server prepends the destination servers to that message. When the message reaches a router, the latter can use the list to decide on the paths that the message should follow, as shown in Fig. 13-7.

Figure 13-7. Naive content-based routing.



Taking this last approach as our starting point, we can refine the capabilities of routers for deciding where to forward messages to. To that end, each server broadcasts its subscription across the network so that routers can compose routing filters. For example, assume that node 3 in Fig. 13-7 subscribes to messages for which an attribute a lies in the range [0,3], but that node 4 wants messages with a ε [2,5]. In this case, router R2 will create a routing filter as a table with an entry for each of its outgoing links (in this case three: one to node 3, one to node 4, and one toward router R1), as shown in Fig. 13-8.
[Page 603]

Figure 13-8. A partially filled routing table.

Interface	Filter
To node 3	a ε [0,3]
To node 4	a ε [2,5]
Toward router R1	(unspecified)

More interesting is what happens at router R1. In this example, the subscriptions from nodes 3 and 4 dictate that any message with a lying in the interval [0,3] ∪ [2,5] = [0,5] should be forwarded along the path to router R2, and this is precisely the information that R1 will store in its table. It is not difficult to imagine that more intricate subscription compositions can be supported.

This simple example also illustrates that whenever a node leaves the system, or when it is no longer interested in specific messages, it should cancel its subscription and essentially broadcast this information to all routers. This cancellation, in turn, may lead to adjusting various routing filters. Late adjustments will at worst lead to unnecessary traffic as messages may be forwarded along paths for which there are no longer subscribers. Nevertheless, timely adjustments are needed to keep performance at an acceptable level.

One of the problems with content-based routing is that although the principle of composing routing filters is simple, identifying the links along which an incoming message must be forwarded can be compute-intensive. The computational complexity comes from the implementation of matching attribute values to subscriptions, which essentially boils down to an entry-by-entry comparison. How this comparison can be done efficiently is described in Carzaniga et al. (2003).

13.4.2. Supporting Composite Subscriptions

The examples so far form relatively simple extensions to routing tables. These extensions suffice when subscriptions take the form of vectors of (attribute, value/range) pairs. However, there is often a need for more sophisticated expressions of subscriptions. For example, it may be convenient to express compositions of subscriptions in which a process specifies in a single subscription that it is interested in very different types of data items. To illustrate, a process may want to see data items on stocks from IBM and data on their revenues, but sending data items of only one kind is not useful.

To handle subscription compositions, Li and Jacobsen (2005) proposed to design routers analogous to rule databases. In effect, subscriptions are transformed into rules stating under which conditions published data should be forwarded, and along which outgoing links. It is not difficult to imagine that this may lead to content-based routing schemes that are far more advanced than the routing filters described above. Supporting subscription composition is strongly related to naming issues in coordination-based systems, which we discuss next.

13.5. Naming

Let us now pay some more attention to naming in coordination-based systems. So far, we have mostly assumed that every published data item has an associated vector of n (attribute, value) pairs and that processes can subscribe to data items by specifying predicates over these attribute values. In general, this naming scheme can be readily applied, although systems differ with respect to attribute types, values, and the predicates that can be used.

For example, with JavaSpaces we saw that essentially only comparison for equality is supported, although this can be relatively easily extended in application-specific ways. Likewise, many commercial publish/subscribe systems support only rather primitive string-comparison operators.

One of the problems we already mentioned is that in many cases we cannot simply assume that every data item is tagged with values for all attributes. In particular, we will see that a data item has only one associated (attribute, value) pair, in which case it is also referred to as an event. Support for subscribing to events, and notably composite events largely dictates the discussion on naming issues in publish/subscribe systems. What we have discussed so far should be considered as the more primitive means for supporting coordination in distributed systems. We now address in more depth events and event composition.

When dealing with composite events, we need to take two different issues into account. The first one is to describe compositions. Such descriptions form the basis for subscriptions. The second issue is how to collect (primitive) events and subsequently match them to subscriptions. Pietzuch et al. (2003) have proposed a general framework for event composition in distributed systems. We take this framework as the basis for our discussion.

13.5.1. Describing Composite Events

Let us first consider some examples of composite events to give a better idea of the complexity that we may need to deal with. Fig. 13-9 shows examples of increasingly complex composite events. In this example, R4.20 could be an air-conditioned and secured computer room.

Figure 13-9. Examples of events in a distributed system.
(This item is displayed on page 605 in the print version)

Ex.	Description
S1	Notify when room R4.20 is unoccupied
S2	Notify when R4.20 is unoccupied and the door is unlocked
S3	Notify when R4.20 is unoccupied for 10 seconds while the door is unlocked

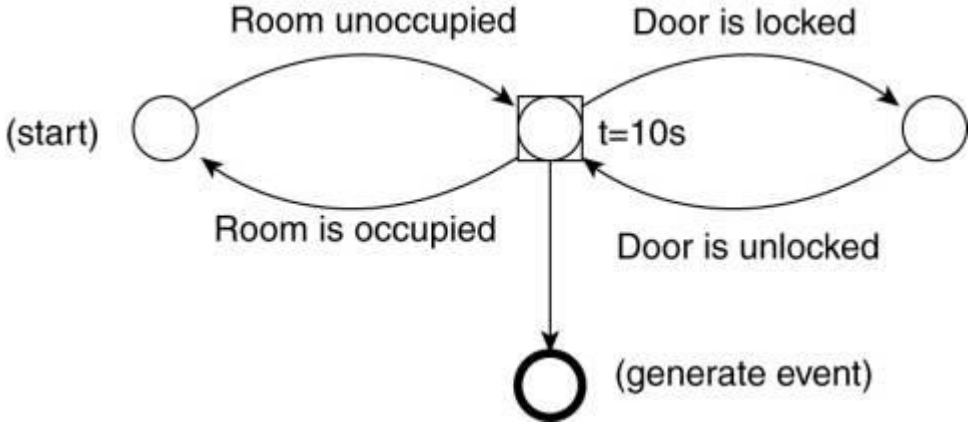
S4	Notify when the temperature in R4.20 rises more than 1 degree per 30 minutes
S5	Notify when the average temperature in R4.20 is more than 20 degrees in the past 30 minutes

The first two subscriptions are relatively easy. S1 is an example that can be handled by a primitive discrete event, whereas S2 is a simple composition of two discrete events. Subscription S3 is more complex as it requires that the system can also report time-related events. Matters are further complicated if subscriptions involve aggregated values required for computing gradients (S4) or averages (S5). Note that in the case of S5 we are requiring a continuous monitoring of the system in order to send notifications on time.
[Page 605]

The basic idea behind an event-composition language for distributed systems is to enable the formulation of subscriptions in terms of primitive events. In their framework, Pietzuch et al. provide a relatively simple language for an extended type of finite-state machine (FSM). The extensions allow for the specification of sojourn times in states, as well as the generation of new (composite) events. The precise details of their language are not important for our discussion here. What is important is that subscriptions can be translated into FSMs.

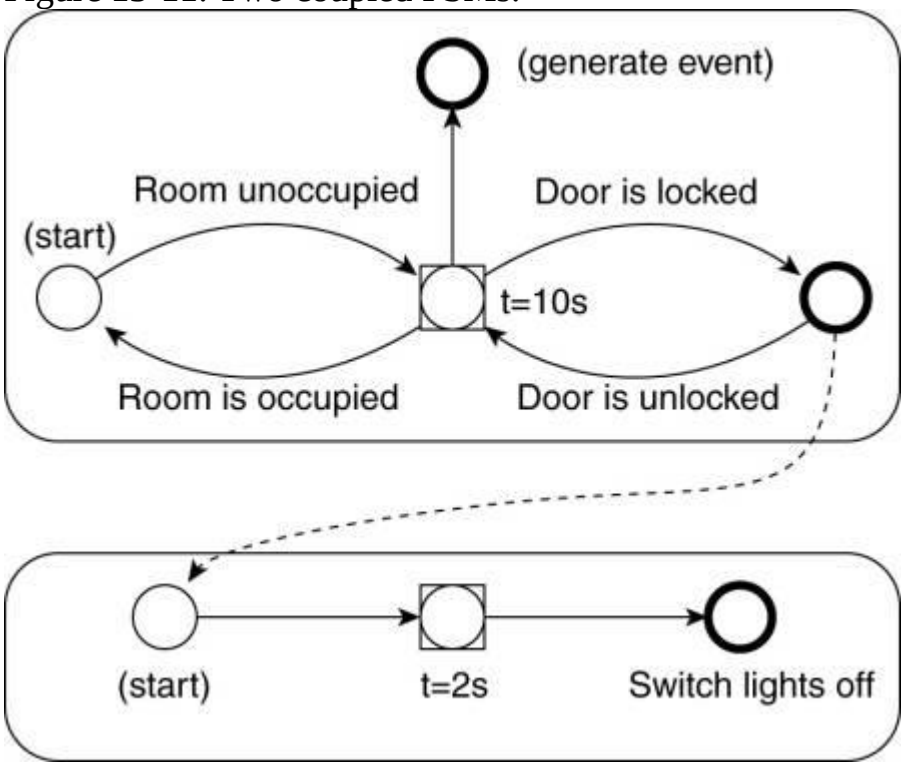
To give an example, Fig. 13-10 shows the FSM for subscription S3 from Fig. 13-9. The special case is given by the timed state, indicated by the label "t = 10s" which specifies that a transition to the final state is made if the door is not locked within 10 seconds.

Figure 13-10. The finite state machine for subscription S3 from Fig. 13-9.



Much more complex subscriptions can be described. An important aspect is that these FSMs can often be decomposed into smaller FSMs that communicate by passing events to each other. Note that such an event communication would normally trigger a state transition at the FSM for which that event is intended. For example, assume that we want to automatically turn off the lights in room R4.20 after 2 seconds when we are certain that nobody is there anymore (and the door is locked). In that case, we can reuse the FSM from Fig. 13-10 if we let it generate an event for a second FSM that will trigger the lighting, as shown in Fig. 13-11 [Page 606]

Figure 13-11. Two coupled FSMs.



The important observation here is that these two FSMs can be implemented as separate processes in the distributed system. In this case, the FSM for controlling the lighting will subscribe to the composed event that is triggered when R4.20 is unoccupied and the door is locked. This leads to distributed detectors which we discuss next.

13.5.2. Matching Events and Subscriptions

Now consider a publish/subscribe system supporting composite events. Every subscription is provided in the form of an expression that can be translated into a finite state machine (FSM). State transitions are essentially triggered by primitive events that take place, such as leaving a room or locking a door.

To match events and subscriptions, we can follow a simple, naive implementation in which every subscriber runs a process implementing the finite state machine associated with its subscription. In that case, all the primitive events that are relevant for a specific subscription will have to be forwarded to the subscriber. Obviously, this will generally not be very efficient.

A much better approach is to consider the complete collection of subscriptions, and decompose subscriptions into communicating finite state machines, such that some of these FSMs are shared between different subscriptions. An example of this sharing was shown in Fig. 13-11. This approach toward handling subscriptions leads to what are known as distributed event detectors. Note that a distribution of event detectors is similar in nature to the distributed resolution of names in various naming systems. Primitive events lead to state transitions in relatively simple finite state machines, in turn triggering the generation of composite events. The latter can then lead to state transitions in other FSMs, again possibly leading to further event generation. Of course, events translate to messages that are sent over the network to processes that subscribed to them.
[Page 607]

Besides optimizing through sharing, breaking down subscriptions into communicating FSMs also has the potential advantage of optimizing network usage. Consider again the events related to monitoring the computer room we described above. Assuming that there only processes interested in the composite events, it makes sense to compose these events close to the computer room. Such a placement will prevent having to send the primitive events across the network. Moreover, when considering Fig. 13-9, we see that we may only need to send the alarm when noticing that the room is unoccupied for 10 seconds while the door is unlocked. Such an event will generally occur rarely in comparison to, for example, (un)locking the door.

Decomposing subscriptions into distributed event detectors, and subsequently optimally placing them across a distributed system is still subject to much research. For example, the last word on subscription languages has not been said, and especially the trade-off between expressiveness and efficiency of implementations will attract a lot of attention. In most cases, the more expressive a language is, the more unlikely there will be an efficient distributed implementation. Current proposals such as by Demers et al. (2006) and by Liu

and Jacobsen (2004) confirm this. It will take some years before we see these techniques being applied to commercial publish/subscribe systems.

13.6. Synchronization

Synchronization in coordination-based systems is generally restricted to systems supporting generative communication. Matters are relatively straightforward when only a single server is used. In that case, processes can be simply blocked until tuples become available, but it is also simpler to remove them. Matters become complicated when the shared dataspace is replicated and distributed across multiple servers, as we describe next.

13.7. Consistency and Replication

Replication plays a key role in the scalability of coordination-based systems, and notably those for generative communication. In the following, we first consider some standard approaches as have been explored in a number of systems such as JavaSpaces. Next, we describe some recent results that allow for the dynamic and automatic placement of tuples depending on their access patterns.
[Page 608]

13.7.1. Static Approaches

The distributed implementation of a system supporting generative communication frequently requires special attention. We concentrate on possible distributed implementations of a JavaSpace server, that is, an implementation by which the collection of tuple instances may be distributed and replicated across several machines. An overview of implementation techniques for tuple-based runtime systems is given by Rowstron (2001).

1. General Considerations

2. An efficient distributed implementation of a JavaSpace has to solve two problems:

3. How to simulate associative addressing without massive searching.

4. How to distribute tuple instances among machines and locate them later.

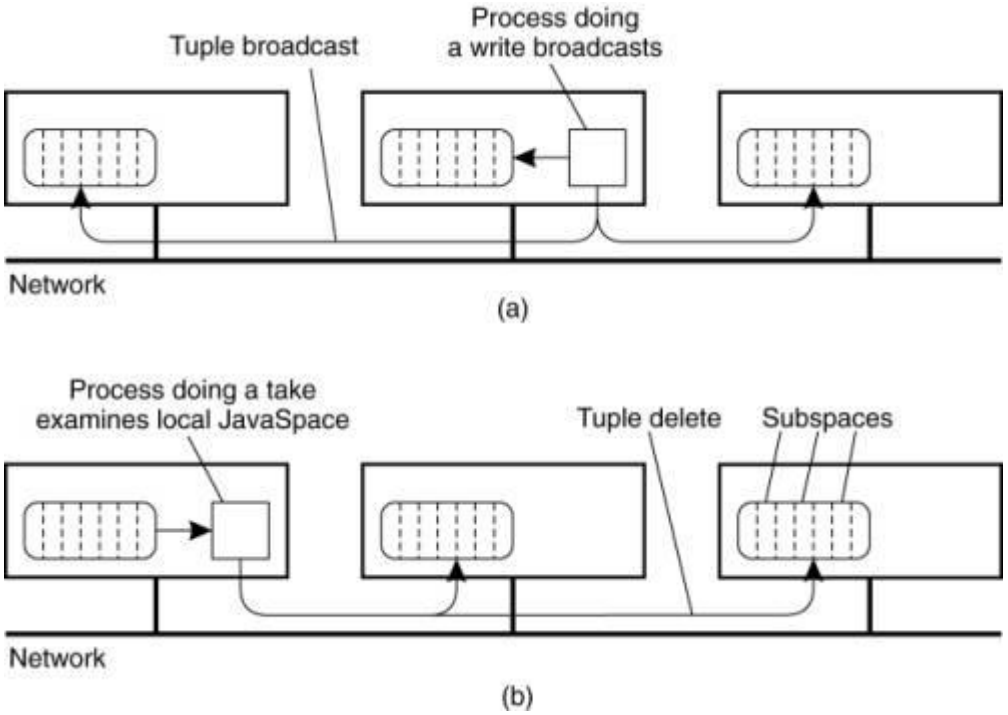
The key to both problems is to observe that each tuple is a typed data structure. Splitting the tuple space into subspaces, each of whose tuples is of the same type simplifies programming and makes certain optimizations possible. For example, because tuples are typed, it becomes possible to determine at compile time which subspace a call to a write, read, or take operates on. This partitioning means that only a fraction of the set of tuple instances has to be searched.

In addition, each subspace can be organized as a hash table using (part of) its i-th tuple field as the hash key. Recall that every field in a tuple instance is a marshaled reference to an object. JavaSpaces does not prescribe how marshaling should be done. Therefore, an implementation may decide to marshal a reference in such a way that the first few bytes are used as an identifier of the type of the object that is being marshaled. A call to a write, read, or take operation can then be executed by computing the hash function of the ith field to find the position in the table where the tuple instance belongs. Knowing the subspace and table position eliminates all searching. Of course, if the ith field of a read or take operation is NULL, hashing is not possible, so a complete search of the subspace is generally needed. By carefully choosing the field to hash on, however, searching can often be avoided.

Additional optimizations are also used. For example, the hashing scheme described above distributes the tuples of a given subspace into bins to restrict searching to a single bin. It is possible to place different bins on different machines, both to spread the load more widely and to take advantage of locality. If the hashing function is the type identifier modulo the number of machines, the number of bins scales linearly with the system size [see also Bjornson (1993)].
[Page 609]

On a network of computers, the best choice depends on the communication architecture. If reliable broadcasting is available, a serious candidate is to replicate all the subspaces in full on all machines, as shown in Fig. 13-12. When a write is done, the new tuple instance is broadcast and entered into the appropriate subspace on each machine. To do a read or take operation, the local subspace is searched. However, since successful completion of a take requires removing the tuple instance from the JavaSpace, a delete protocol is required to remove it from all machines. To prevent race conditions and deadlocks, a two-phase commit protocol can be used.

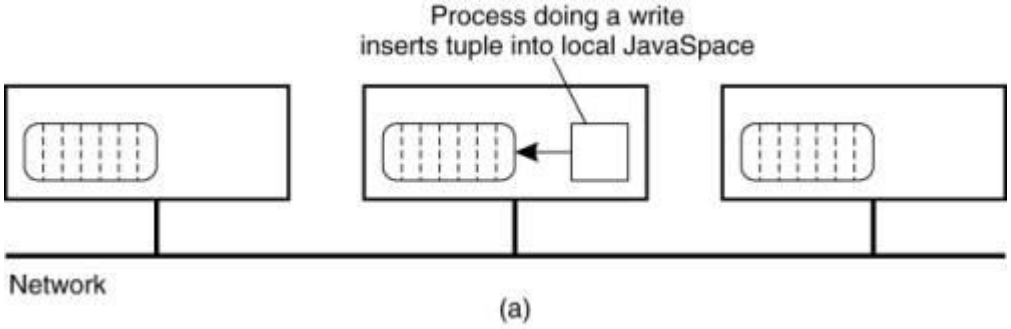
Figure 13-12. A JavaSpace can be replicated on all machines. The dotted lines show the partitioning of the JavaSpace into subspaces. (a) Tuples are broadcast on write. (b) reads are local, but the removing an instance when calling take must be broadcast.

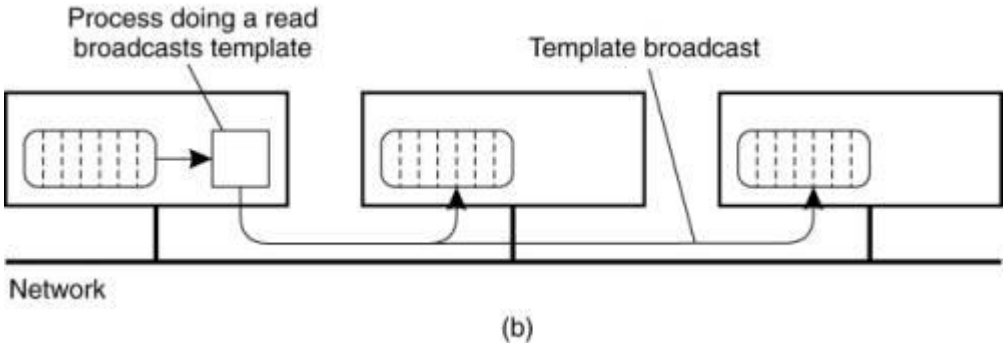


This design is straightforward, but may not scale well as the system grows in the number of tuple instances and the size of the network. For example, implementing this scheme across a wide-area network is prohibitively expensive.

The inverse design is to do writes locally, storing the tuple instance only on the machine that generated it, as shown in Fig. 13-13. To do a read or take, a process must broadcast the template tuple. Each recipient then checks to see if it has a match, sending back a reply if it does.

Figure 13-13. Nonreplicated JavaSpace. (a) A write is done locally. (b) A read or take requires the template tuple to be broadcast in order to find a tuple instance. (This item is displayed on page 610 in the print version)



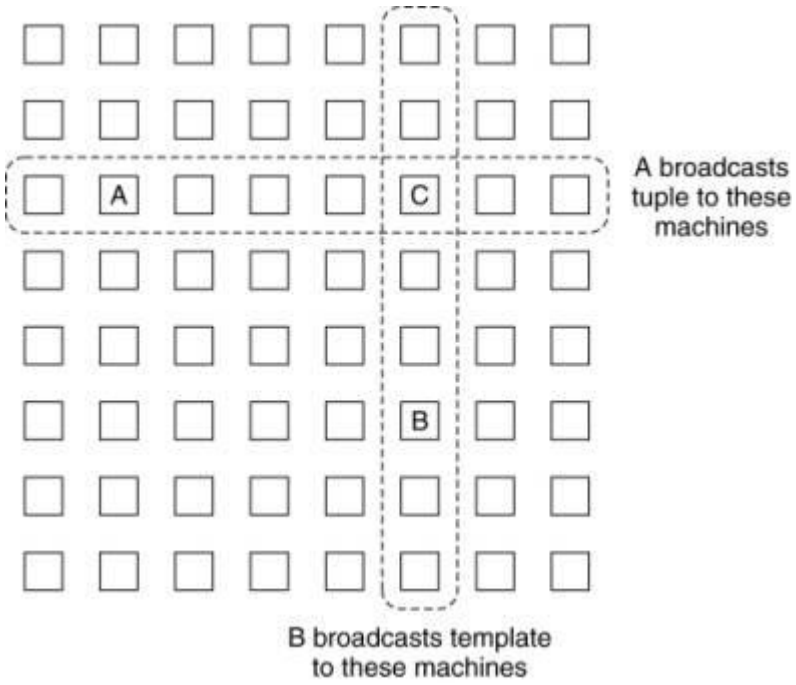


(b)

If the tuple instance is not present, or if the broadcast is not received at the machine holding the tuple, the requesting machine retransmits the broadcast request ad infinitum, increasing the interval between broadcasts until a suitable tuple instance materializes and the request can be satisfied. If two or more tuple instances are sent, they are treated like local writes and the instances are effectively moved from the machines that had them to the one doing the request. In fact, the runtime system can even move tuples around on its own to balance the load. Carriero and Gelernter (1986) used this method for implementing the Linda tuple space on a LAN.
[Page 610]

These two methods can be combined to produce a system with partial replication. As a simple example, imagine that all the machines logically form a rectangular grid, as shown in Fig. 13-14. When a process on a machine A wants to do a write, it broadcasts (or sends by point-to-point message) the tuple to all machines in its row of the grid. When a process on a machine B wants to read or take a tuple instance, it broadcasts the template tuple to all machines in its column. Due to the geometry, there will always be exactly one machine that sees both the tuple instance and the template tuple (C in this example), and that machine makes the match and sends the tuple instance to the process requesting for it. This approach is similar to using quorum-based replication as we discussed in Chap. 7.

Figure 13-14. Partial broadcasting of tuples and template tuples.
(This item is displayed on page 611 in the print version)



The implementations we have discussed so far have serious scalability problems caused by the fact that multicasting is needed either to insert a tuple into a tuple space, or to remove one. Wide-area implementations of tuple spaces do not exist. At best, several different tuple spaces can coexist in a single system, where each tuple space itself is implemented on a single server or on a local-area network. This approach is used, for example, in PageSpaces (Ciancarini et al., 1998) and WCL (Rowstron and Wray, 1998). In WCL, each tuple-space server is responsible for an entire tuple space. In other words, a process will always be directed to exactly one server. However, it is possible to migrate a tuple space to a different server to enhance performance. How to develop an efficient wide-area implementation of tuple spaces is still an open question.
[Page 611]

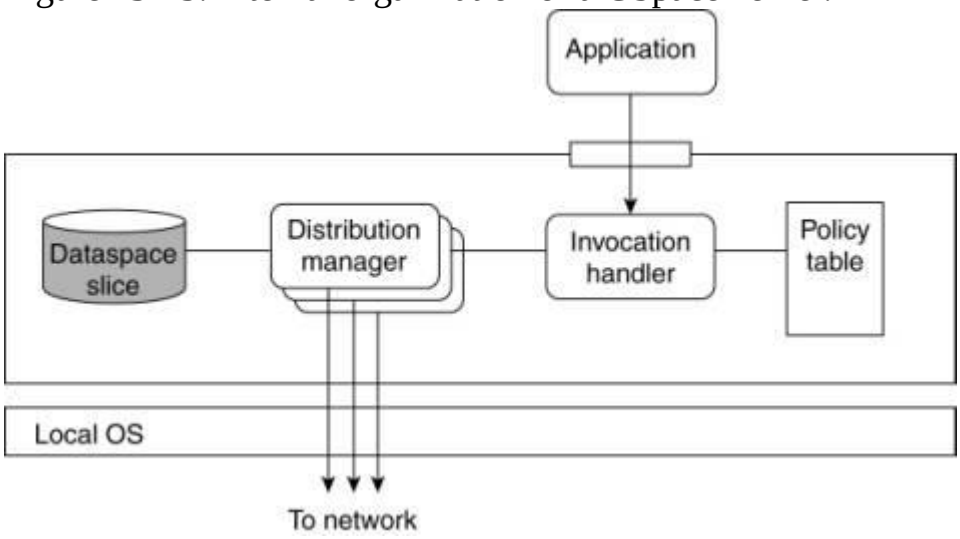
13.7.2. Dynamic Replication

Replication in coordination-based systems has generally been restricted to static policies for parallel applications like those discussed above. In commercial applications, we also see relatively simple schemes in which entire dataspaces or otherwise statically predefined parts of a data set are subject to a single policy (GigaSpaces, 2005). Inspired by the fine-grained replication of Web documents in Globule, performance improvements can also be achieved when differentiating replication between the different kinds of data stored in a dataspace. This differentiation is supported by GSpace, which we briefly discuss in this section.

GSpace Overview

GSpace is a distributed coordination-based system that is built on top of Java-Spaces (Russello et al., 2004, 2006). Distribution and replication of tuples in GSpace is done for two different reasons: improving performance and availability. A key element in this approach is the separation of concerns: tuples that need to be replicated for availability may need to follow a different strategy than those for which performance is at stake. For this reason, the architecture of GSpace has been set up to support a variety of replication policies, and such that different tuples may follow different policies.
[Page 612]

Figure 13-15. Internal organization of a GSpace kernel.



The principal working is relatively simple. Every application is offered an interface with a read, write, and take interface, similar to what is offered by Java-Spaces. However, every call is picked up by a local invocation handler which looks up the policy that should be followed for the specific call. A policy is selected based on the type and content of the tuple/template that is passed as part of the call. Every policy is identified by a template, similar to the way that templates are used to select tuples in other Java-based shared dataspaces as we discussed previously.

The result of this selection is a reference to a distribution manager, which implements the same interface, but now does it according to a specific replication policy. For example, if a master/slave policy has been implemented, a read operation may be implemented by immediately reading a tuple from the locally

available dataspace. Likewise, a write operation may require that the distribution manager forwards the update to the master node and awaits an acknowledgment before performing the operation locally.

Finally, every GSpace kernel has a local dataspace, called a slice, which is implemented as a full-fledged, nondistributed version of JavaSpaces.

In this architecture (of which some components are not shown for clarity), policy descriptors can be added at runtime, and likewise, distribution managers can be changed as well. This setup allows for a fine-grained tuning of the distribution and replication of tuples, and as is shown in Russello et al. (2004), such fine-tuning allows for much higher performance than is achievable with any fixed, global strategy that is applied to all tuples in a dataspace.
[Page 613]

Adaptive Replication

However, the most important aspect with systems such as GSpace is that replication management is automated. In other words, rather than letting the application developer figure out which combination of policies is the best, it is better to let the system monitor access patterns and behavior and subsequently adopt policies as necessary.

To this end, GSpace follows the same approach as in Globule: it continuously measures consumed network bandwidth, latency, and memory usage and depending on which of these metrics is considered most important, places tuples on different nodes and chooses the most appropriate way to keep replicas consistent. The evaluation of which policy is the best for a given tuple is done by means of a central coordinator which simply collects traces from the nodes that constitute the GSpace system.

An interesting aspect is that from time to time we may need to switch from one replication policy to another. There are several ways in which such a transition can take place. As GSpace aims to separate mechanisms from policies as best as possible, it can also handle different transition policies. The default case is to temporarily freeze all operations for a specific type of tuple, remove all replicas and reinsert the tuple into the shared dataspace but now following the newly selected replication policy. However, depending on the new replication policy, a different way of making the transition may be possible (and cheaper). For example, when switching from no replication to master/slave replication, one approach could be to lazily copy tuples to the slaves when they are first accessed.

13.8. Fault Tolerance

When considering that fault tolerance is fundamental to any distributed system, it is somewhat surprising how relatively little attention has been paid to fault tolerance in coordination-based systems, including basic publish/subscribe systems as well as those supporting generative communication. In most cases, attention focuses on ensuring efficient reliability of data delivery, which essentially boils down to guaranteeing reliable communication. When the middleware is also expected to store data items, as is the case with generative communication, some effort is paid to reliable storage. Let us take a closer look at these two cases.

13.8.1. Reliable Publish-Subscribe Communication

In coordination-based systems where published data items are matched only against live subscribers, reliable communication plays a crucial role. In this case, fault tolerance is most often implemented through the implementation of reliable multicast systems that underly the actual publish/subscribe software. There are several issues that are generally taken care of. First, independent of the way that content-based routing takes place, a reliable multicast channel is set up. Second, process fault tolerance needs to be handled. Let us take a look how these matters are addressed in TIB/Rendezvous.
[Page 614]

Example: Fault Tolerance in TIB/Rendezvous

TIB/Rendezvous assumes that the communication facilities of the underlying network are inherently unreliable. To compensate for this unreliability, whenever a rendezvous daemon publishes a message to other daemons, it will keep that message for at least 60 seconds. When publishing a message, a daemon attaches a (subject independent) sequence number to that message. A receiving daemon can detect it is missing a message by looking at sequence numbers (recall that messages are delivered to all daemons). When a message has been missed, the publishing daemon is requested to retransmit the message.

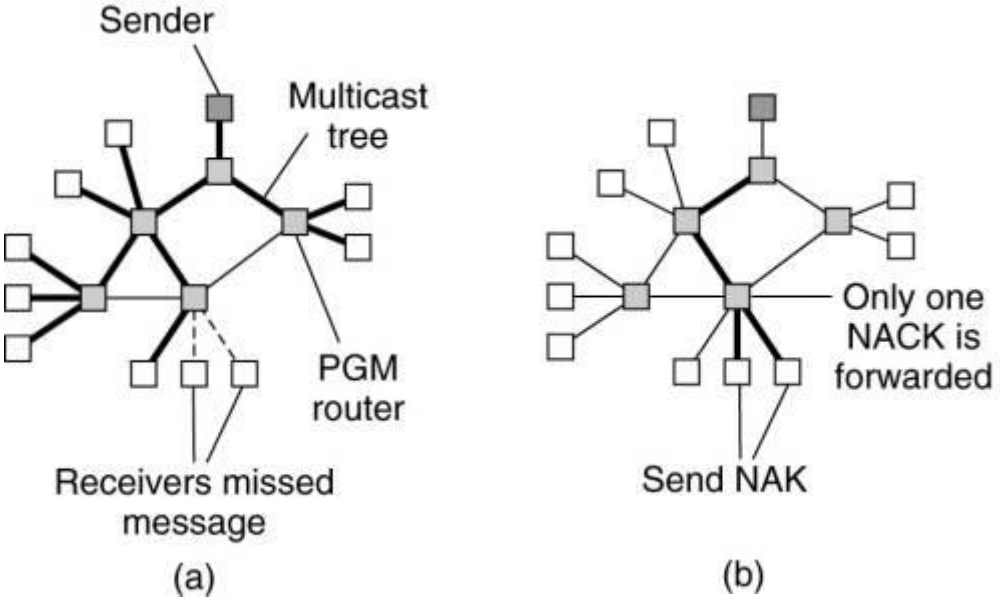
This form of reliable communication cannot prevent that messages may still be lost. For example, if a receiving daemon requests a retransmission of a message that has been published more than 60 seconds ago, the publishing daemon will generally not be able to help recover this lost message. Under normal circumstances, the publishing and subscribing applications will be notified that a

communication error has occurred. Error handling is then left to the applications to deal with.

Much of the reliability of communication in TIB/Rendezvous is based on the reliability offered by the underlying network. TIB/Rendezvous also provides reliable multicasting using (unreliable) IP multicasting as its underlying communication means. The scheme followed in TIB/Rendezvous is a transport-level multicast protocol known as Pragmatic General Multicast (PGM), which is described in Speakman et al. (2001). We will discuss PGM briefly.

PGM does not provide hard guarantees that when a message is multicast it will eventually be delivered to each receiver. Fig. 13-16(a) shows a situation in which a message has been multicast along a tree, but it has not been delivered to two receivers. PGM relies on receivers detecting that they have missed messages for which they will send a retransmission request (i.e., a NAK) to the sender. This request is sent along the reverse path in the multicast tree rooted at the sender, as shown in Fig. 13-16(b). Whenever a retransmission request reaches an intermediate node, that node may possibly have cached the requested message, at which point it will handle the retransmission. Otherwise, the node simply forwards the NAK to the next node toward the sender. The sender is ultimately responsible for retransmitting a message.

Figure 13-16. The principle of PGM. (a) A message is sent along a multicast tree. (b) A router will pass only a single NAK for each message. (c) A message is retransmitted only to receivers that have asked for it.
(This item is displayed on page 615 in the print version)



PGM takes several measures to provide a scalable solution to reliable multicasting. First, if an intermediate node receives several retransmission requests for exactly the same message, only one retransmission request is forwarded toward the sender. In this way, an attempt is made to ensure that only a single NAK reaches the sender, so that a feedback implosion is avoided. We already came across this problem in Chap. 8 when discussing scalability issues in reliable multicasting.
[Page 615]

A second measure taken by PGM is to remember the path through which a NAK traverses from receivers to the sender, as is shown in Fig. 13-16(c). When the sender finally retransmits the requested message, PGM takes care that the message is multicast only to those receivers that had requested retransmission. Consequently, receivers to which the message had been successfully delivered are not bothered by retransmissions for which they have no use.

Besides the basic reliability scheme and reliable multicasting through PGM, TIB/Rendezvous provides further reliability by means of certified message delivery. In this case, a process uses a special communication channel for sending or receiving messages. The channel has an associated facility, called a ledger, for keeping track of sent and received certified messages. A process that wants to receive certified messages registers itself with the sender of such messages. In effect, registration allows the channel to handle further reliability issues for which the rendezvous daemons provide no support. Most of these issues are hidden from applications and are handled by the channel's implementation.

When a ledger is implemented as a file, it becomes possible to provide reliable message delivery even in the presence of process failures. For example, when a receiving process crashes, all messages it misses until it recovers again are stored in a sender's ledger. Upon recovery, the receiver simply contacts the ledger and requests the missed messages to be retransmitted.

To enable the masking of process failures, TIB/Rendezvous provides a simple means to automatically activate or deactivate processes. In this context, an active process normally responds to all incoming messages, while an inactive one does not. An inactive process is a running process that can handle only special events as we explain shortly.
[Page 616]

Processes can be organized into a group, with each process having a unique rank associated with it. The rank of a process is determined by its (manually assigned) weight, but no two processes in the same group may have the same rank. For each

group, TIB/Rendezvous will attempt to have a group-specific number of processes active, called the group's active goal. In many cases, the active goal is set to one so that all communication with a group reduces to a primary-based protocol as discussed in Chap. 7.

An active process regularly sends a message to all other members in the group to announce that it is still up and running. Whenever such a heartbeat message is missing, the middleware will automatically activate the highest-ranked process that is currently inactive. Activation is accomplished by a callback to an action operation that each group member is expected to implement. Likewise, when a previously crashed process recovers again and becomes active, the lowest-ranked currently active process will be automatically deactivated.

To keep consistent with the active processes, special measures need to be taken by an inactive process before it can become active. A simple approach is to let an inactive process subscribe to the same messages as any other group member. An incoming message is processed as usual, but no reactions are ever published. Note that this scheme is akin to active replication.

13.8.2. Fault Tolerance in Shared Dataspaces

When dealing with generative communication, matters become more complicated. As also noted in Tolksdorf and Rowstron (2000), as soon as fault tolerance needs to be incorporated in shared dataspaces, solutions can often become so inefficient that only centralized implementations are feasible. In such cases, traditional solutions are applied, notably using a central server that is backed up in using a simple primary-backup protocol, in combination with checkpointing.

An alternative is to deploy replication more aggressively by placing copies of data items across the various machines. This approach has been adopted in GSpace, essentially deploying the same mechanisms it uses for improving performance through replication. To this end, each node computes its availability, which is then used in computing the availability of a single (replicated) data item (Russello et al., 2006).

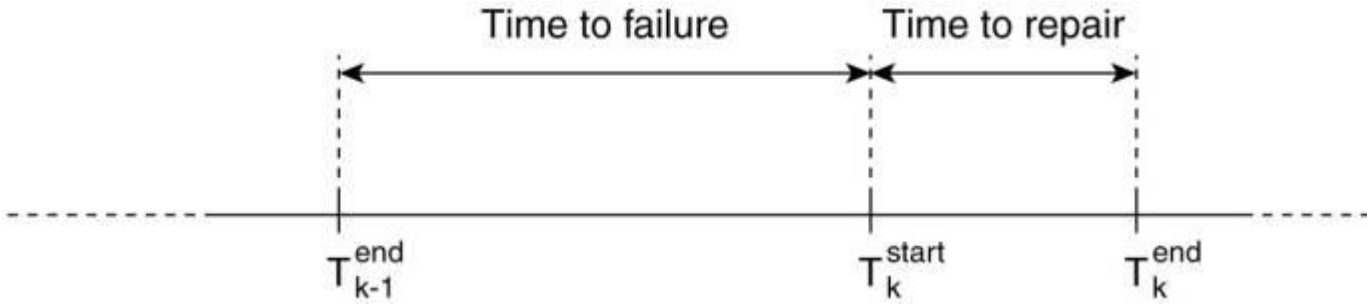
To compute its availability, a node regularly writes a timestamp to persistent storage, allowing it to compute the time when it is up, and the time when it was down. More precisely, availability is computed in terms of the mean time to failure (MTTF) and the mean time to repair (MTTR):

$$Availability\ node = \frac{MTTF}{MTTF+MTTR}$$

To compute MTTF and MTTR, a node simply looks at the logged timestamps, as shown in Fig. 13-17. This will allow it to compute the averages for the time between failures, leading to an availability of:

$$Availability\ node = \frac{\sum_{k=1}^{n} (T_k^{start} - T_{k-1}^{end})}{\sum_{k=1}^{n} (T_k^{start} - T_{k-1}^{end}) + \sum_{k=1}^{n} (T_k^{end} - T_k^{start})}$$

Figure 13-17. The time line of a node experiencing failures.



Note that it is necessary to regularly log timestamps and that T_k^{start} can be taken only as a best estimate of when a crash occurred. However, the thus computed availability will be pessimistic, as the actual time that a node crashed for the kth time will be slightly later than T_k^{start}. Also, instead of taking averages since the beginning, it is also possible to take only the last N crashes into account.

In GSpace, each type of data item has an associated primary node that is responsible for computing that type's availability. Given that a data item is replicated across m nodes, its availability is computed by considering the availability ai of each of the m nodes leading to:

$$Availability\ data\ item = 1 - \prod_{k=1}^{m} (1 - a_i)$$

By simply taking the availability of a data item into account, as well as those of all nodes, the primary can compute an optimal placement for a data item that will satisfy the availability requirements for a data item. In addition, it can also take other factors into account, such as bandwidth usage and CPU loads. Note that placement may change over time if these factors fluctuate.

13.9. Security

Security in coordination-based systems poses a difficult problem. On the one hand we have stated that processes should be referentially decoupled, but on the other hand we should also ensure the integrity and confidentiality of data. This security is normally implemented through secure (multicast) channels, which effectively require that senders and receivers can authenticate each other. Such authentication violates referential decoupling.
[Page 618]

To solve this problem there are different approaches. One common approach is to set up a network of brokers that handle the processing of data and subscriptions. Client processes will then contact the brokers, who then take care of authentication and authorization. Note that such an approach does require that the clients trust the brokers. However, as we shall see later, by differentiating between types of brokers, it is not necessary that a client has to trust all brokers comprising the system.

By nature of data coordination, authorization naturally translates to confidentiality issues. We will now take a closer look at these issues, following the discussion as presented in Wang et al. (2002).

13.9.1. Confidentiality

One important difference between many distributed systems and coordination-based ones is that in order to provide efficiency, the middleware needs to inspect the content of published data. Without being able to do so, the middleware can essentially only flood data to all potential subscribers. This poses the problem of information confidentiality which refers to the fact that it is sometimes important to disallow the middleware to inspect published data. This problem can be circumvented through end-to-end encryption; the routing substrate only sees source and destination addresses.

If published data items are structured in the sense that every item contains multiple fields, it is possible to deploy partial secrecy. For example, data

regarding real estate may need to be shipped between agents of the same office with branches at different locations, but without revealing the exact address of the property. To allow for content-based routing, the address field could be encrypted, while the description of the property could be published in the clear. To this end, Khurana and Koleva (2006) propose to use a per-field encryption scheme as introduced in Bertino and Ferrari (2002). In this case, the agents belonging to the same branch would share the secret key for decrypting the address field. Of course, this violates referential decoupling, but we will discuss a potential solution to this problem later.

More problematic is the case when none of the fields may be disclosed to the middleware in plaintext. The only solution that remains is that content-based routing takes place on the encrypted data. As routers get to see only encrypted data, possibly on a per-field basis, subscriptions will need to be encoded in such a way that partial matching can take place. Note that a partial match is the basis that a router uses to decide which outgoing link a published data item should be forwarded on.

This problem comes very close to querying and searching through encrypted data, something clearly next to impossible to achieve. As it turns out, maintaining a high degree of secrecy while still offering reasonable performance is known to be very difficult (Kantarcioglu and Clifton, 2005). One of the problems is that if per-field encryption is used, it becomes much easier to find out what the data is all about.
[Page 619]

Having to work on encrypted data also brings up the issue of subscription confidentiality, which refers to the fact that subscriptions may not be disclosed to the middleware either. In the case of subject-based addressing schemes, one solution is to simply use per-field encryption and apply matching on a strict field-by-field basis. Partial matching can be accommodated in the case of compound keywords, which can be represented as encrypted sets of their constituents. A subscriber would then send encrypted forms of such constituents and let the routers check for set membership, as also suggested by Raiciu and Rosenblum (2005). As it turns out, it is even possible to support range queries, provided an efficient scheme can be devised for representing intervals. A potential solution is discussed in Li et al. (2004a).

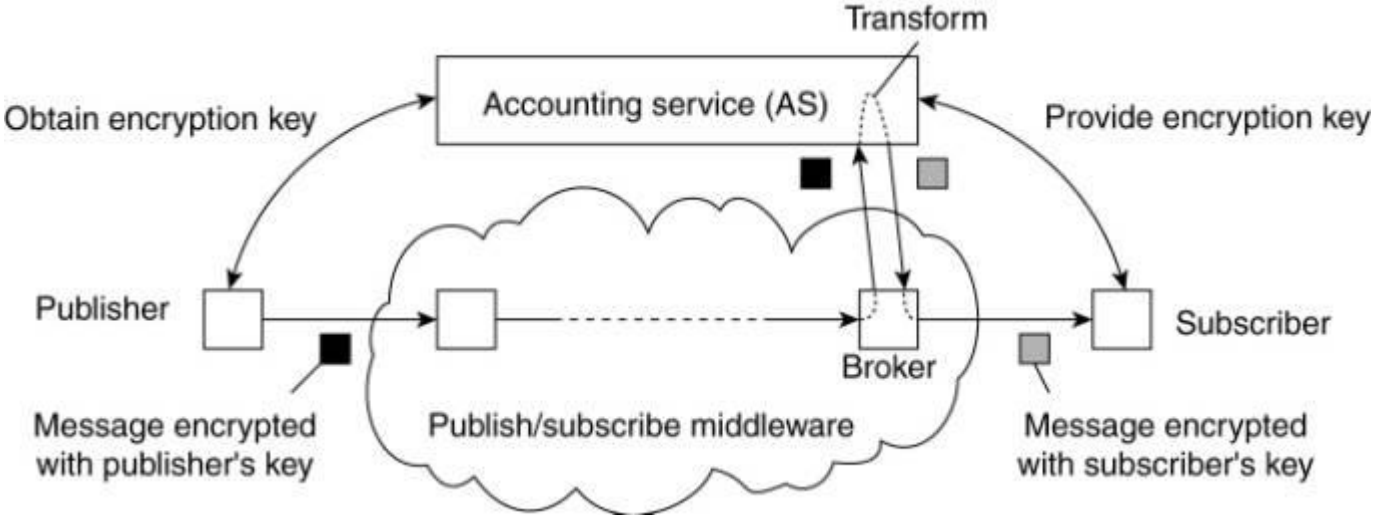
Finally, publication confidentiality is also an issue. In this case, we are touching upon the more traditional access control mechanisms in which certain processes should not even be allowed to see certain messages. In such cases, publishers may want to explicitly restrict the group of possible subscribers. In many cases, this control can be exerted out-of-band at the level of the publishing and subscribing

applications. However, it may convenient that the middleware offers a service to handle such access control.

Decoupling Publishers from Subscribers

If it is necessary to protect data and subscriptions from the middleware, Khurana and Koleva (2006) propose to make use of a special accounting service (AS), which essentially sits between clients (publishers and subscribers) and the actual publish/subscribe middleware. The basic idea is to decouple publishers from subscribers while still providing information confidentiality. In their scheme, subscribers register their interest in specific data items, which are subsequently routed as usual. The data items are assumed to contain fields that have been encrypted. To allow for decryption, once a message should be delivered to a subscriber, the router passes it to the accounting service where it is transformed into a message that only the subscriber can decrypt. This scheme is shown in Fig. 13-18.

Figure 13-18. Decoupling publishers from subscribers using an additional trusted service.
(This item is displayed on page 620 in the print version)



A publisher registers itself at any node of the publish/subscribe network, that is, at a broker. The broker forwards the registration information to the accounting service which then generates a public key to be used by the publisher, and which is signed by the AS. Of course, the AS keeps the associated private key to itself. When a subscriber registers, it provides an encryption key that is forwarded by the broker. It is necessary to go through a separate authentication phase to ensure

that only legitimate subscribers register. For example, brokers should generally not be allowed to subscribe for published data.
[Page 620]

Ignoring many details, when a data item is published, its critical fields will have been encrypted by the publisher. When the data item arrives at a broker who wishes to pass it on to a subscriber, the former requests the AS to transform the message by first decrypting it, and then encrypt it with the key provided by the subscriber. In this way, the brokers will never get to know about content that should be kept secret, while at the same time, publishers and subscribers need not share key information.

Of course, it is crucial that accounting service itself can scale. Various measures can be taken, but one reasonable approach is to introduce realms in a similar way that Kerberos does. In this case, messages in transmission may need to be transformed by re-encrypting them using the public key of a foreign accounting service. For details, we refer the interested reader to (Khurana and Koleva, 2006).

13.9.2. Secure Shared Dataspaces

Very little work has been done when it comes to making shared dataspaces secure. A common approach is to simply encrypt the fields of data items and let matching take place only when decryption succeeds and content matches with a subscription. This approach is described in Vitek et al. (2003). One of the major problems with this approach is that keys may need to be shared between publishers and subscribers, or that the decryption keys of the publishers should be known to authorized subscribers.

Of course, if the shared dataspace is trusted (i.e., the processes implementing the dataspace are allowed to see the content of tuples), matters become much simpler. Considering that most implementations make use of only a single server, extending that server with authentication and authorization mechanisms is often the approach followed in practice.