DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 13
Distributed Coordination-Based Systems

Coordination-based Systems

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.

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 multi-threaded) 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.

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.

Introduction To Coordination Models

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.

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

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

Introduction to Coordination-based Systems

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.

Introduction to Coordination-based Systems
Publish/Subscribe

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.

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.

Introduction to Coordination-based Systems
Linda Tuples

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 Gelemter (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.

Introduction to Coordination-based Systems

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).

Architectures
Overall Approach

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.

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.

Architectures
Overall Approach

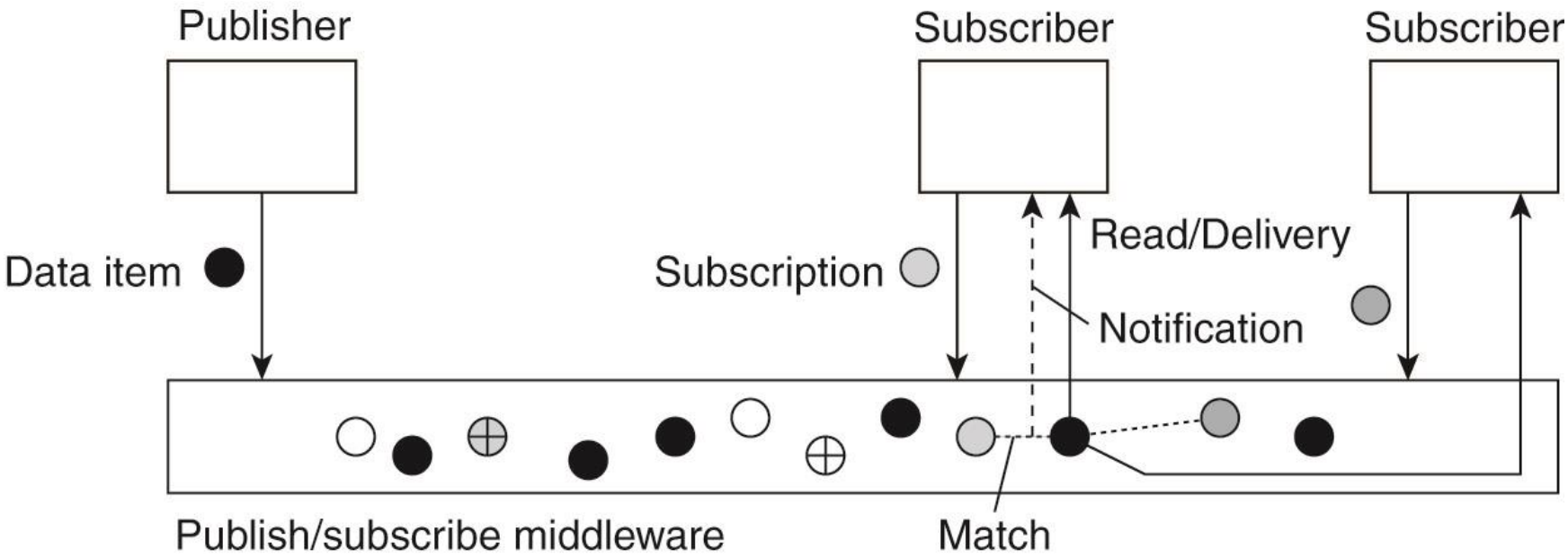

Figure 13-2. The principle of exchanging data items between publishers and subscribers.

Architectures
Overall Approach

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 a_1, ... ,a_n that is used to describe data items. In particular, each published data item is assumed to have an associated vector *(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.

Architectures
Overall Approach

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.

Architectures
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.

Traditional Architectures
Example: Jini and JavaSpaces

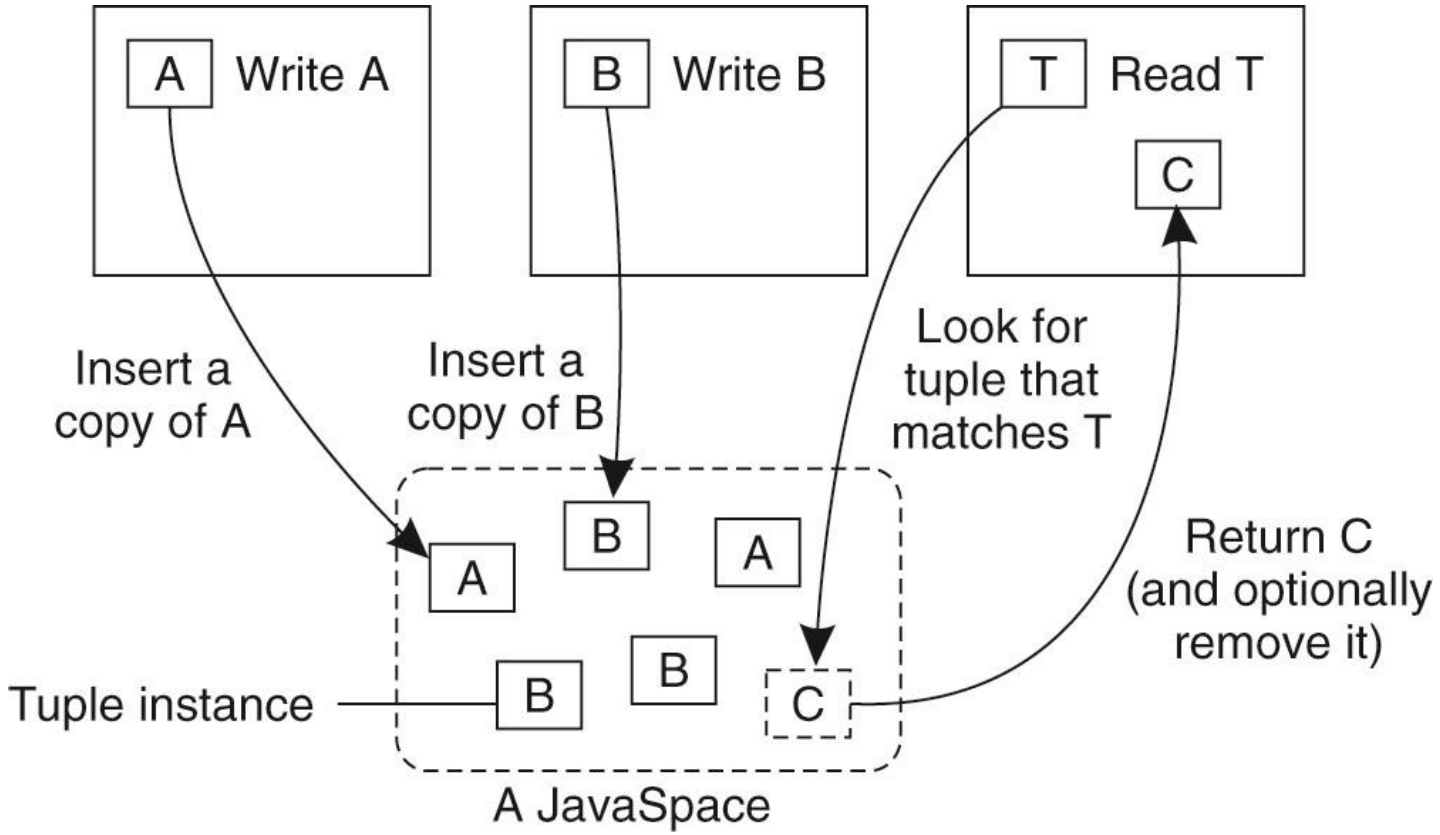

Figure 13-3. The general organization of a JavaSpace in Jini.

Example: TIB/Rendezvous

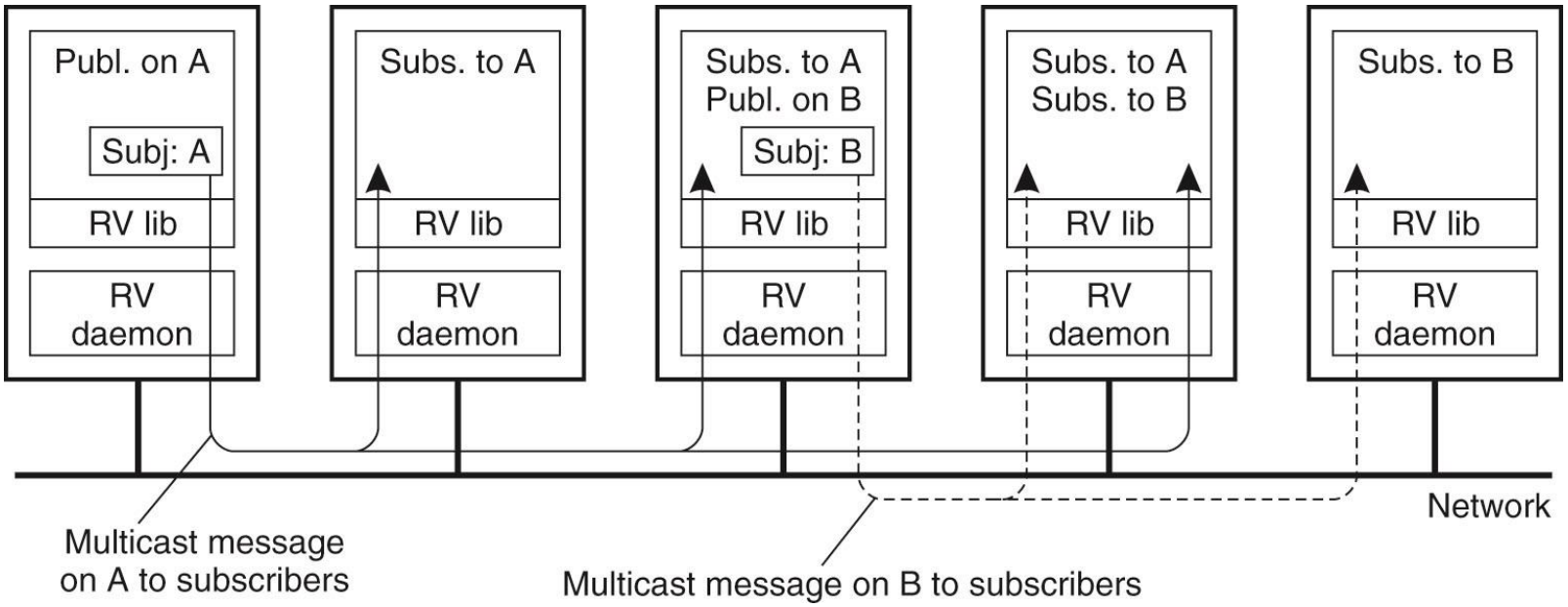

Figure 13-4. The principle of a publish/subscribe system as implemented in TIB/Rendezvous.

Example: A Gossip-Based Publish/Subscribe System

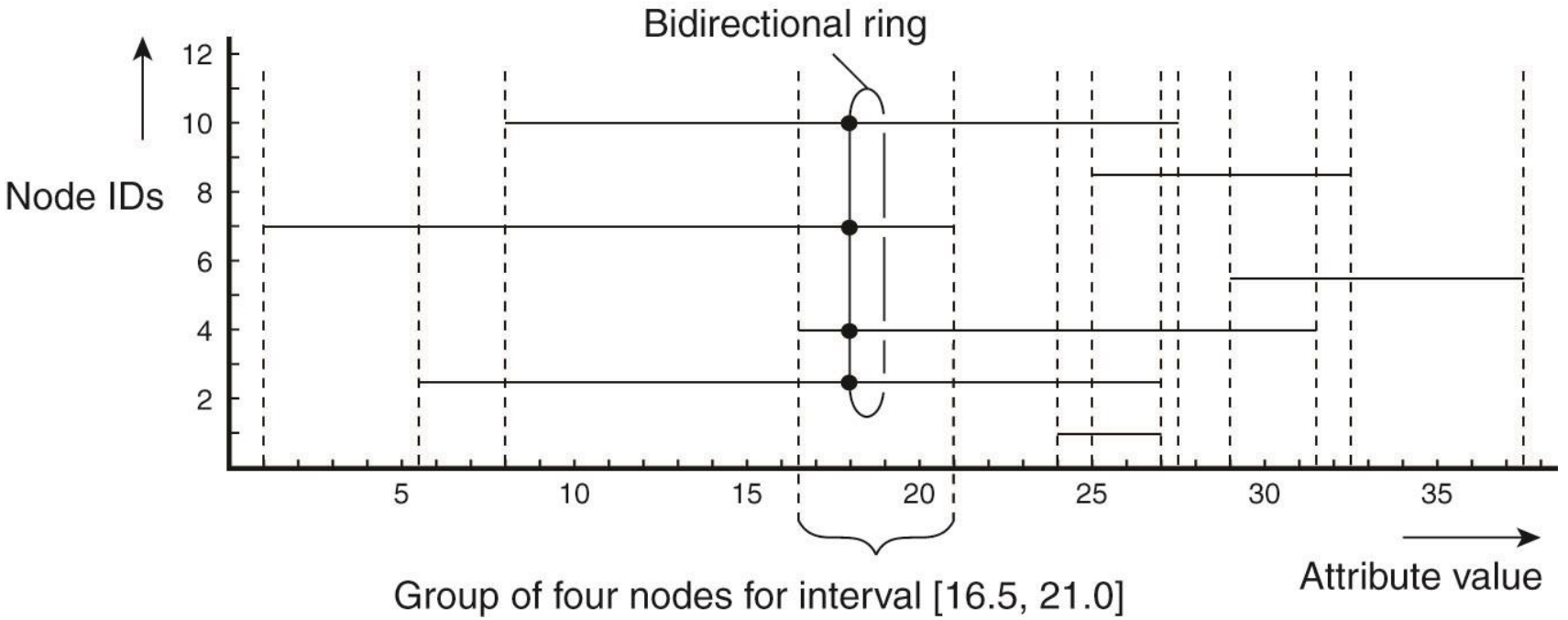

Figure 13-5. Grouping nodes for supporting range queries in a peer-to-peer publish/subscribe system.

Example: Lime

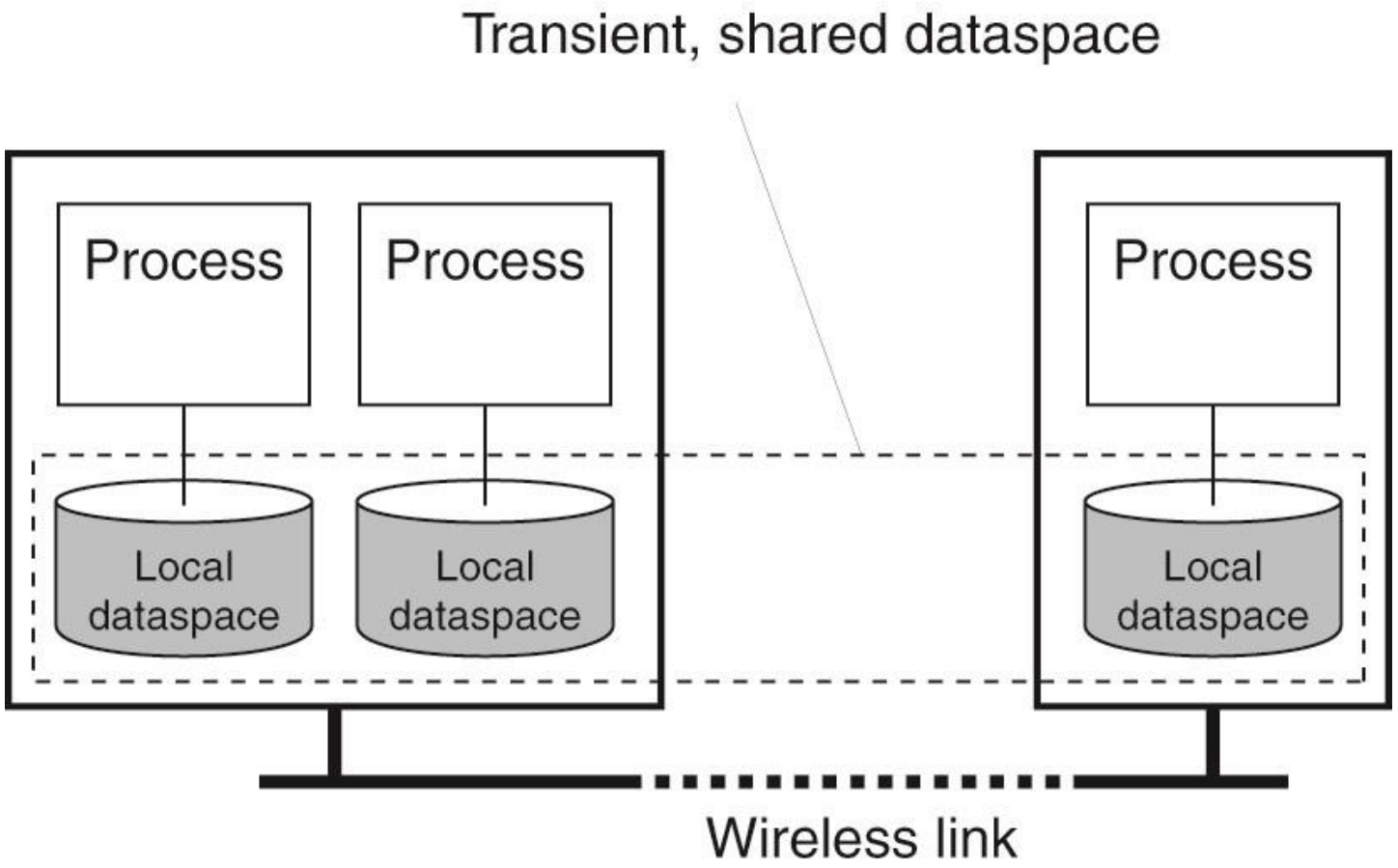

Figure 13-6. Transient sharing of local dataspaces in Lime.

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 (RMI).

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.

An solution is to deploy content-based routing.

COMMUNICATION

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.

COMMUNICATION

Content-Based Routing

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.

COMMUNICATION

Content-Based Routing

Carzaniga et a1. 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.

COMMUNICATION

Content-Based Routing

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 (non-compound) 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.

COMMUNICATION
Content-Based Routing

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.

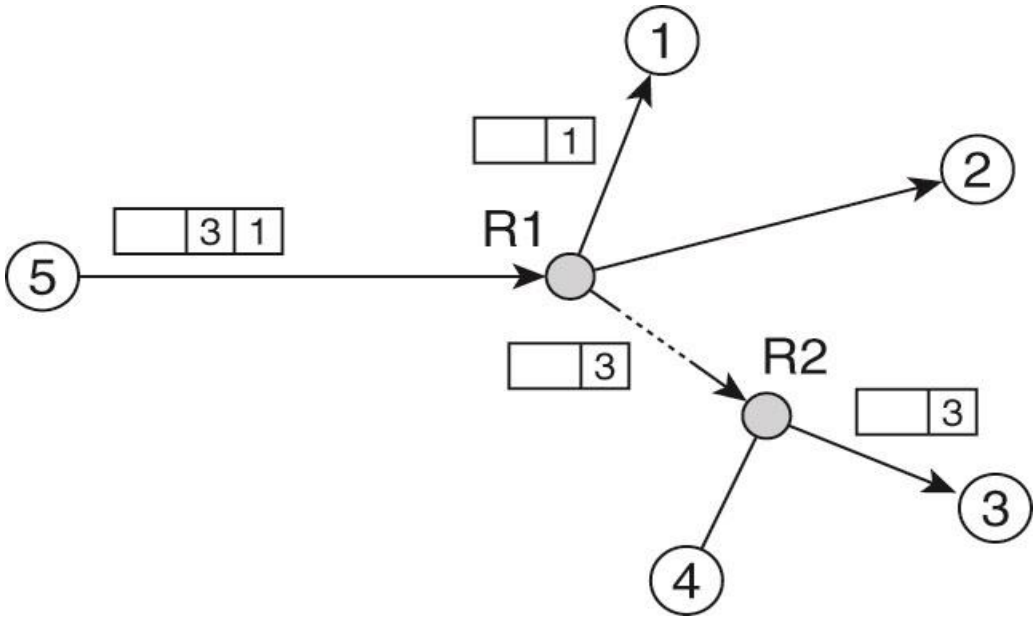

Figure 13-7. Naive content-based routing.

COMMUNICATION
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* in [2,5]. In this case, router *R1* 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 *R* 1), as shown in Fig. 13-8.

Interface	Filter
To node 3	$a \in [0,3]$
To node 4	$a \in [2,5]$
Toward router R_1	(unspecified)

Figure 13-8. A partially filled routing table.

Describing Composite Events (1)

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

Figure 13-9. Examples of events in a distributed system.

Describing Composite Events (2)

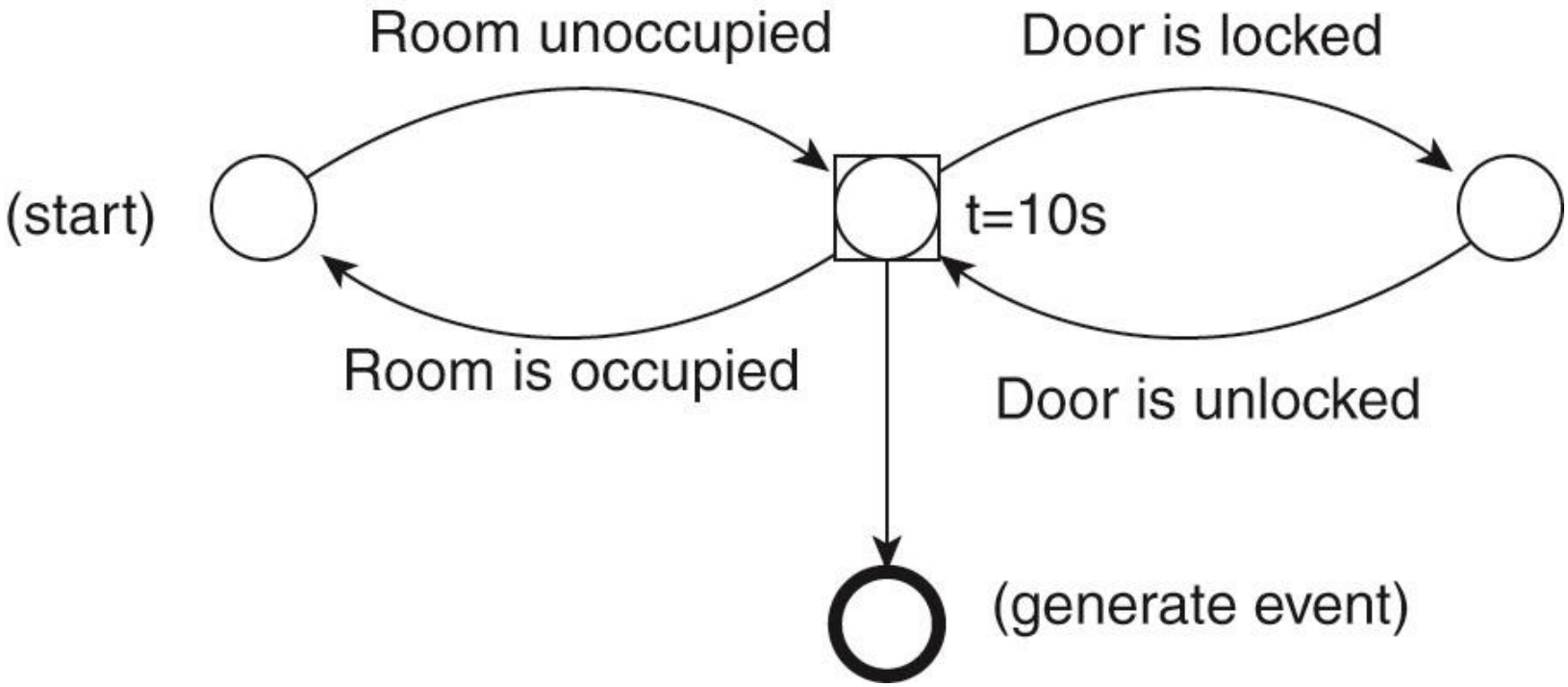

Figure 13-10. The finite state machine for subscription S3 from Fig. 13-9.

Describing Composite Events (3)

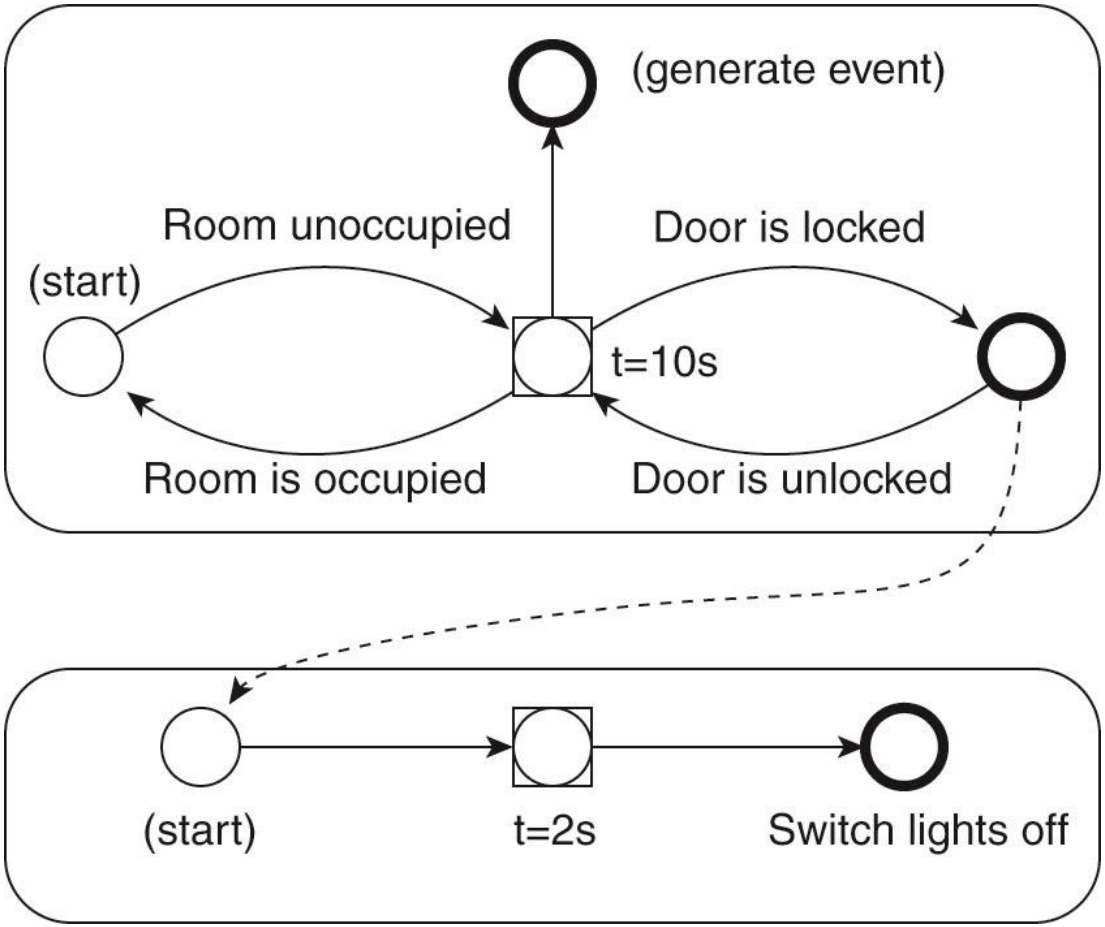

Figure 13-11. Two coupled FSMs.

General Considerations to Static Approaches (1)

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

1. How to simulate associative addressing without massive searching.
2. How to distribute tuple instances among machines and locate them later.

General Considerations to Static Approaches (2)

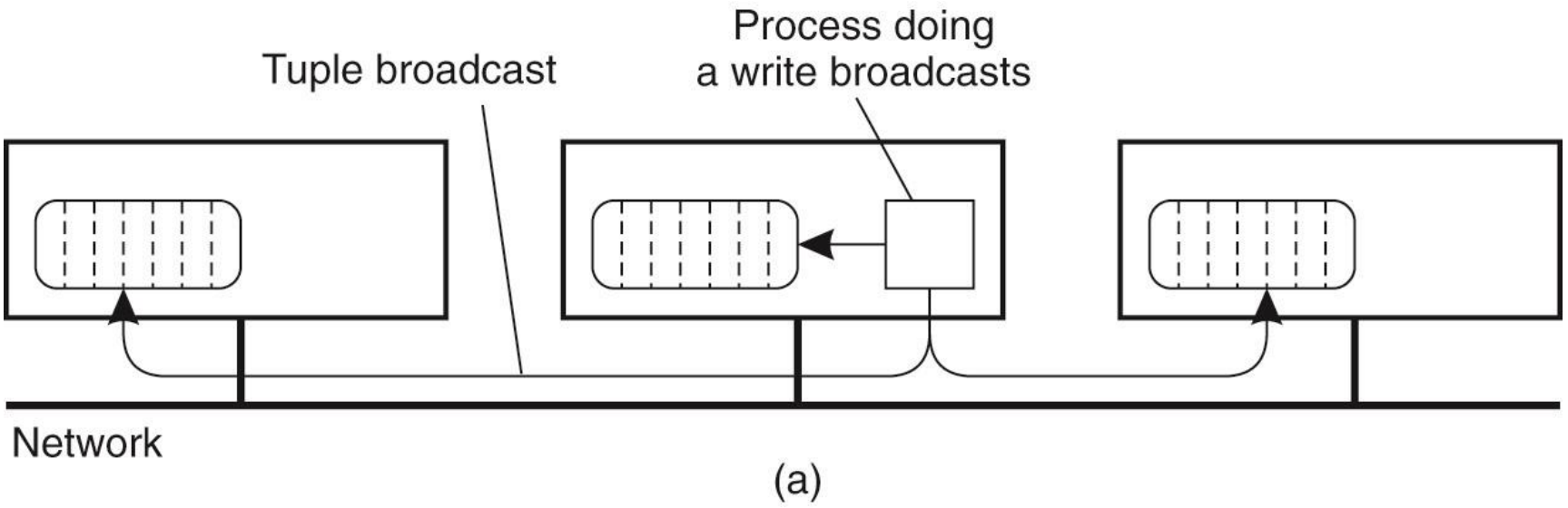

Figure 13-12. A JavaSpace can be replicated on all machines. The dotted lines show the partitioning of the JavaSpace into subspaces. (a) Tuples are broadcast on write.

General Considerations to Static Approaches (3)

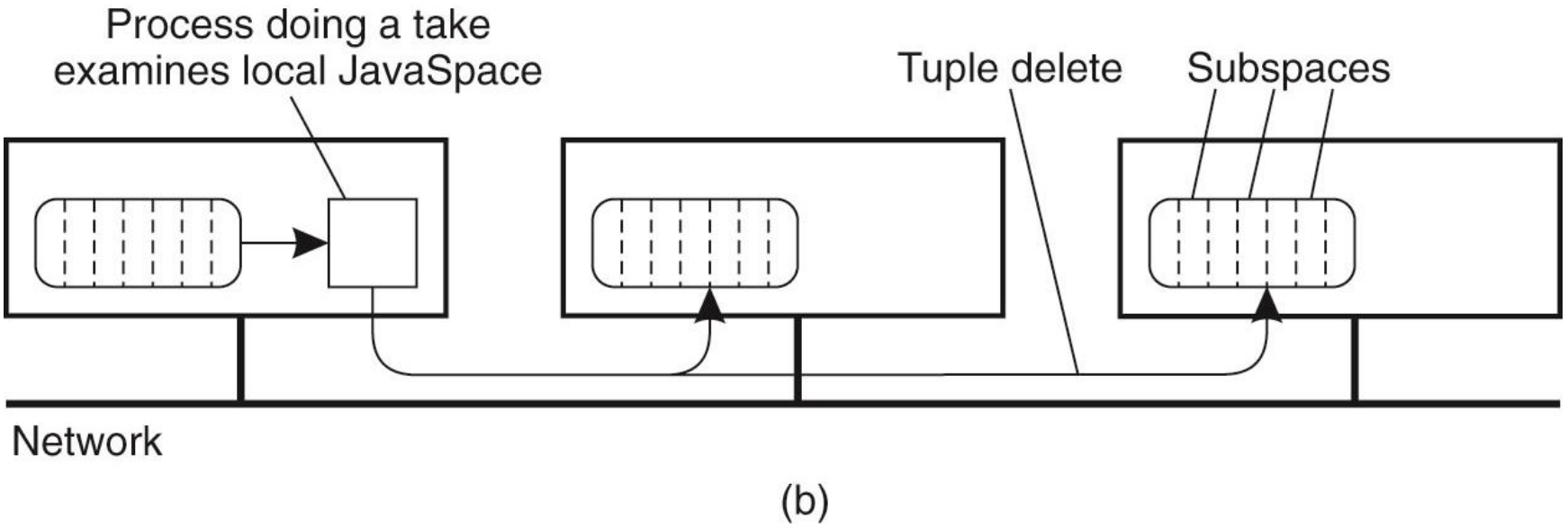

Figure 13-12. A JavaSpace can be replicated on all machines. The dotted lines show the partitioning of the JavaSpace into subspaces. (b) reads are local, but the removing an instance when calling take must be broadcast.

General Considerations to Static Approaches (4)

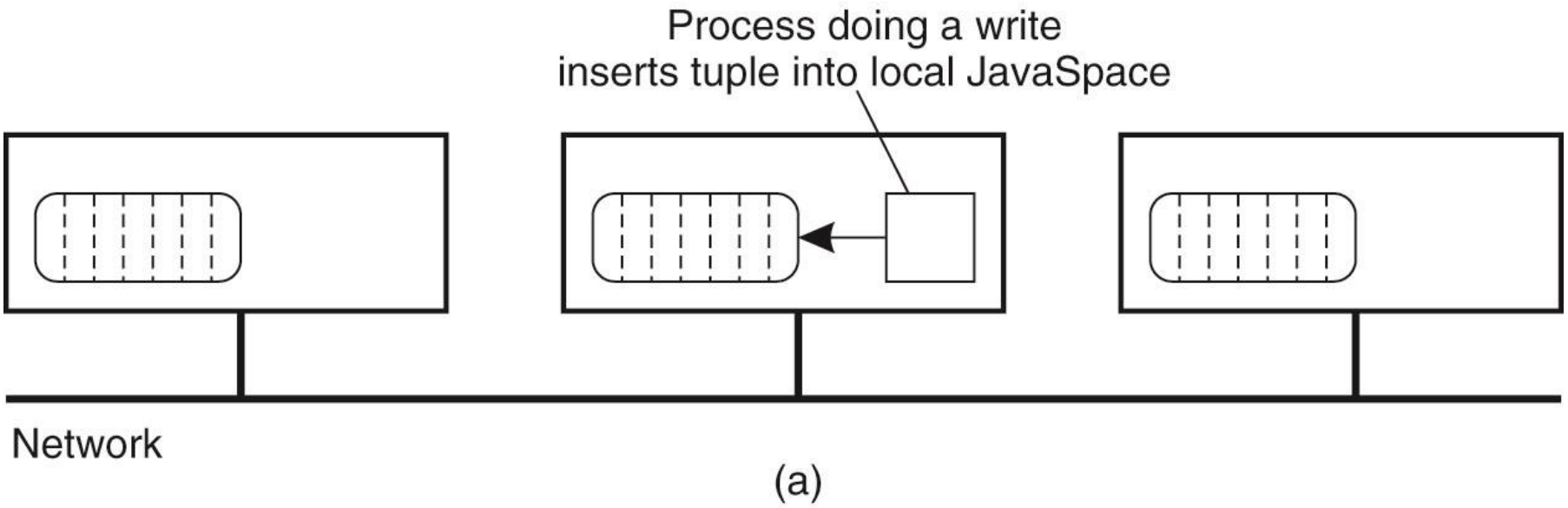

Figure 13-13. Nonreplicated JavaSpace.
(a) A write is done locally.

General Considerations to Static Approaches (5)

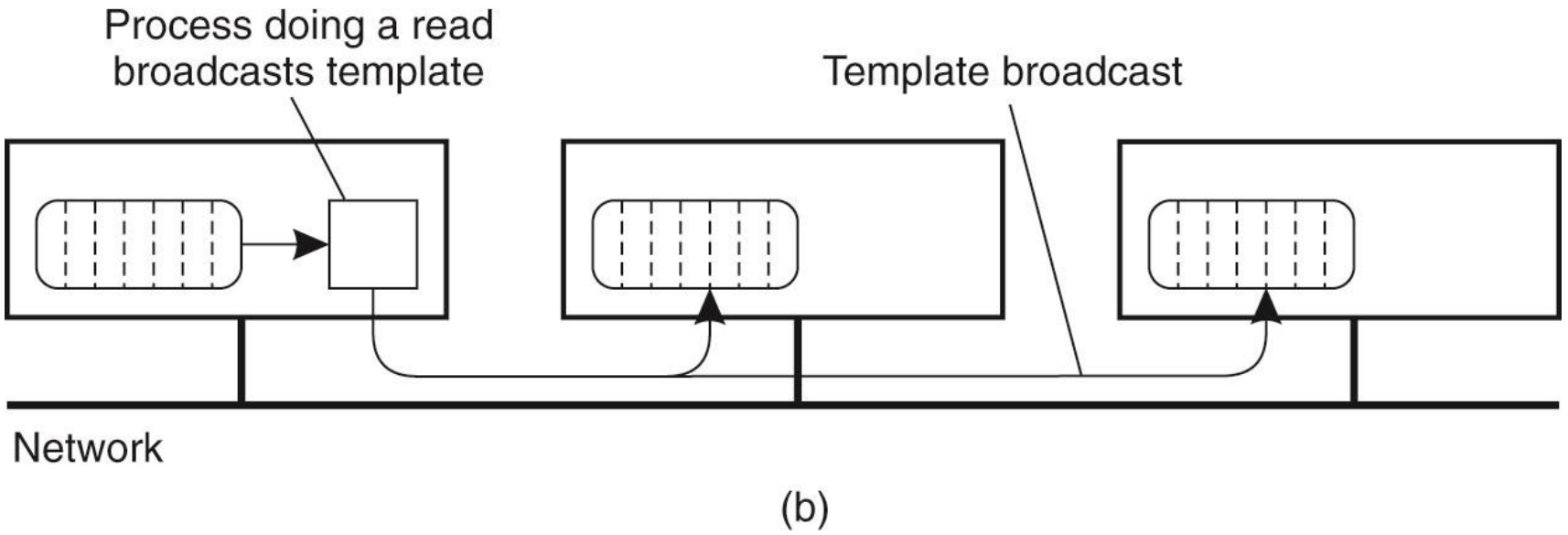

Figure 13-13. Nonreplicated JavaSpace. (b) A read or take requires the template tuple to be broadcast in order to find a tuple instance.

General Considerations to Static Approaches (6)

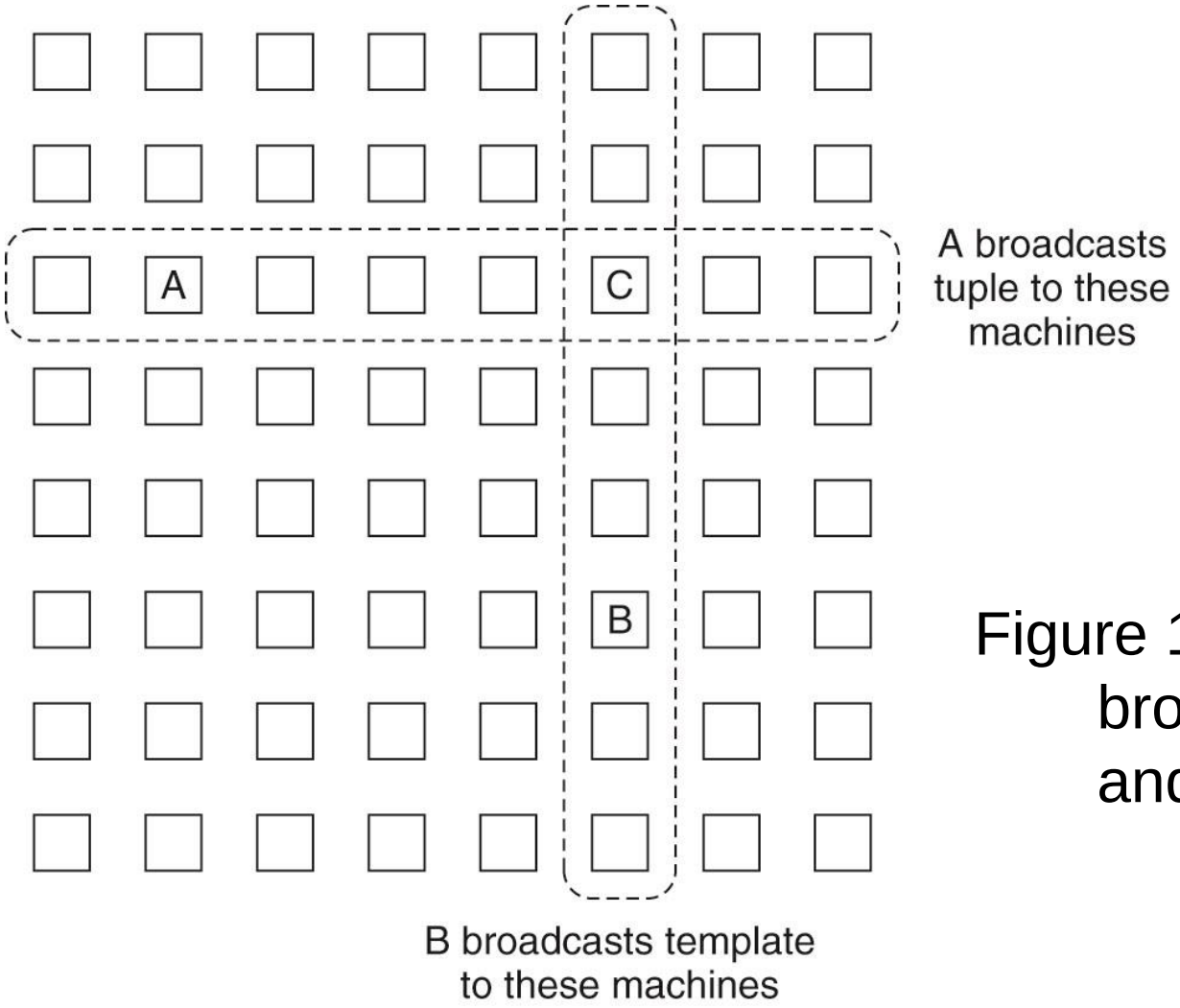

Figure 13-14. Partial broadcasting of tuples and template tuples.

GSpace Overview

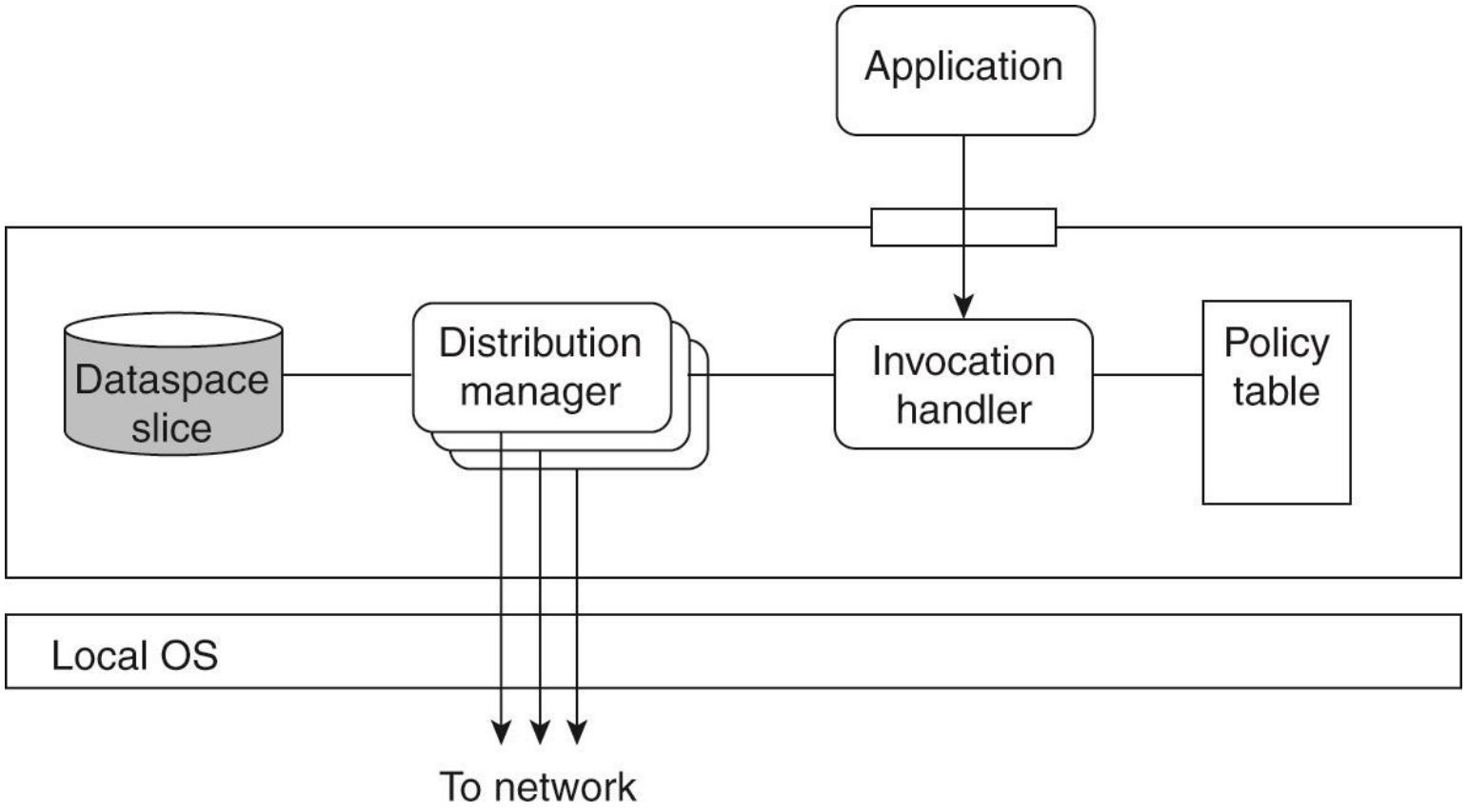

Figure 13-15. Internal organization of a GSpace kernel.

Example: Fault Tolerance in TIB/Rendezvous (1)

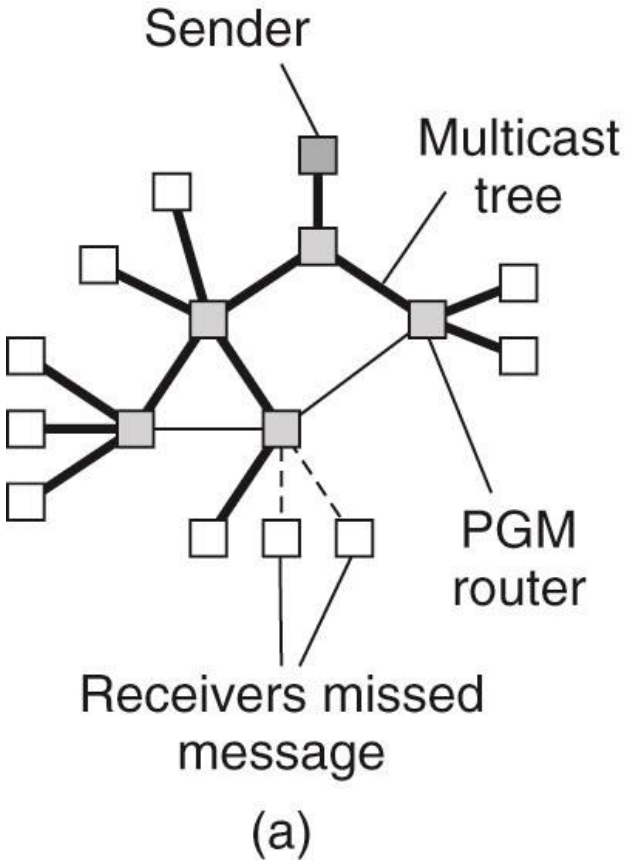

Figure 13-16. The principle of PGM.
(a) A message is sent along a multicast tree.

Example: Fault Tolerance in TIB/Rendezvous (2)

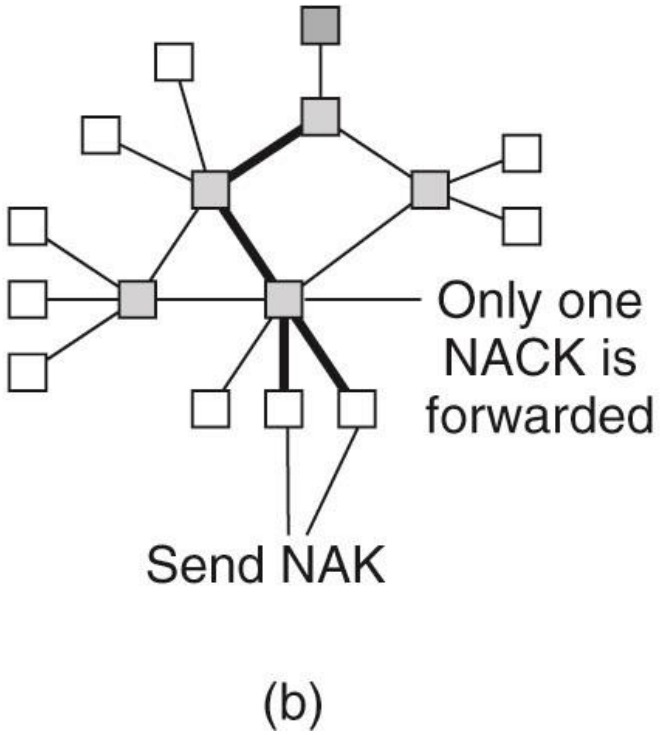

Figure 13-16. The principle of PGM. (b) A router will pass only a single NAK for each message.

Example: Fault Tolerance in TIB/Rendezvous (3)

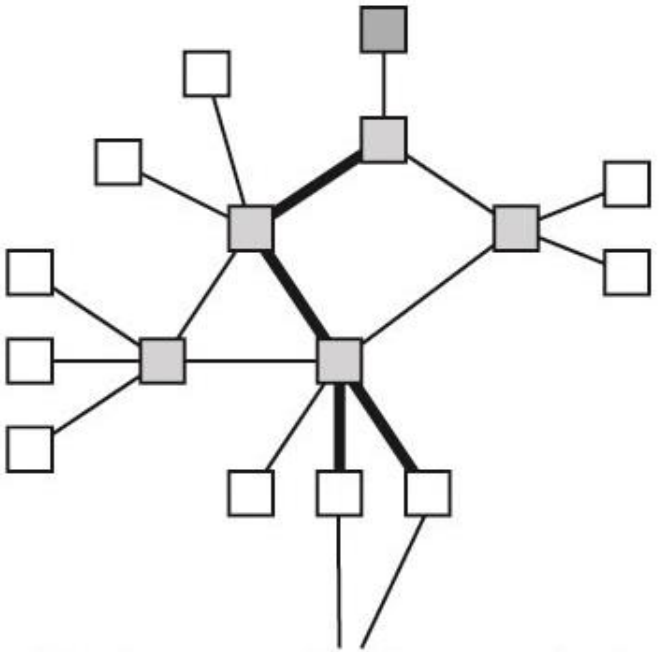

Figure 13-16. The principle of PGM. (c) A message is retransmitted only to receivers that have asked for it

Fault Tolerance in Shared Dataspaces

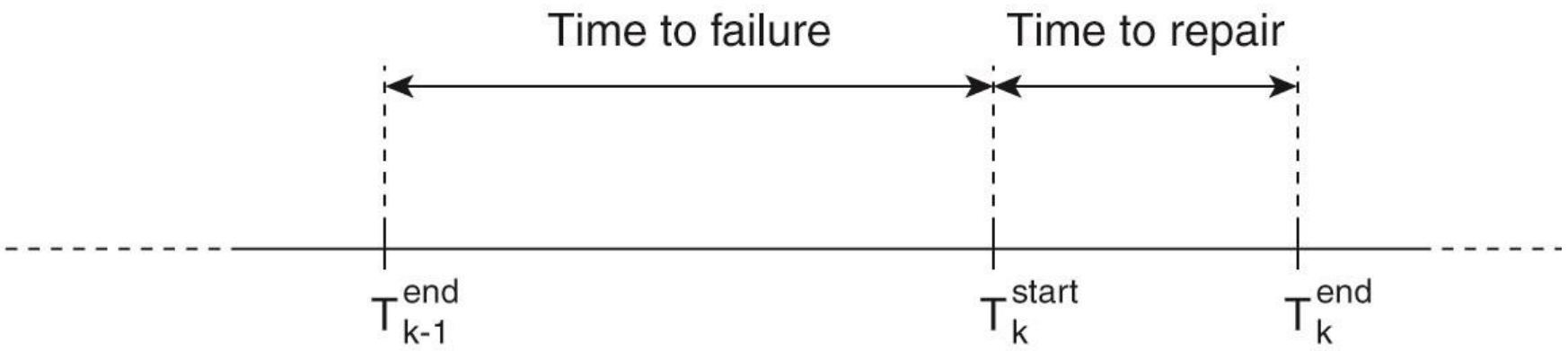

Figure 13-17. The time line of a node experiencing failures.

Decoupling Publishers from Subscribers

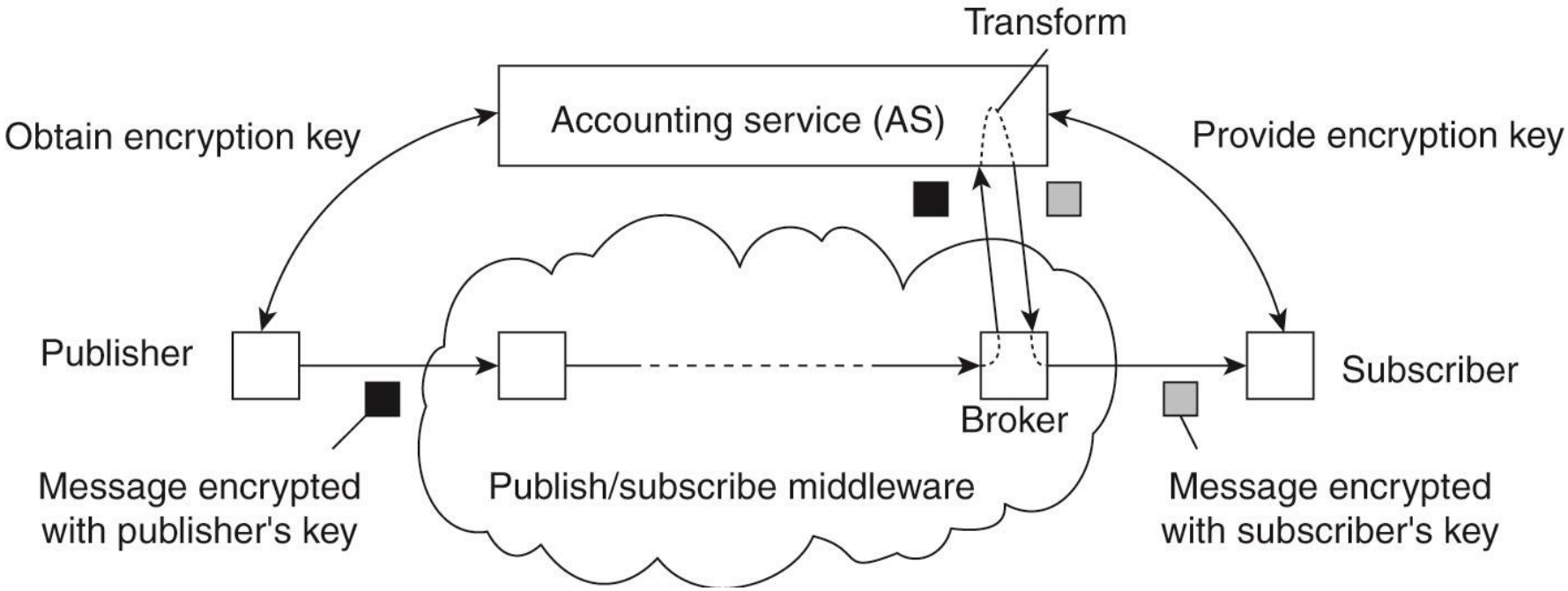

Figure 13-18. Decoupling publishers from subscribers using an additional trusted service.