Slides for Chapter 6:
Indirect Communication

From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems:
Concepts and Design

Edition 5, © Addison-Wesley 2012

Overview of Chapter

Introduction to Indirect Communication

Group communication

Publish-subscribe systems

Message queues

Shared memory approaches

Introduction to indirect communication

Definition of Indirect Communication:

- Communication between entities in a distributed system through an *intermediary*
- No direct coupling between sender and receiver(s)
- Previous chapters covered communication via direct coupling
- Cannot replace server with equivalent one

Two key properties:

- Space uncoupling: sender need not know identity of receivers; senders or receivers may be replaced, updated, replicated, or migrated
- Time uncoupling: sender and receiver do not have to exist at the same time

Introduction to indirect communication

Uses of Indirect Communication:

- Mobile environments, where entities may be connected and disconnected at various times
- Event dissemination, such as event feeds in financial systems
- Main disadvantage: performance overhead

Most systems are either:

- Space and time coupled: most techniques in chapters 4 and 5
- Space and time uncoupled: most techniques in chapter 6
- IP multicast (and some publish-subscribe methods) is space-uncoupled but time-coupled
- Time uncoupling different from asynchronous communication

Figure 6.1
Space and time coupling in distributed systems

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 15.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Overview of Chapter

Introduction to Indirect Communication

Group communication

Publish-subscribe systems

Message queues

Shared memory approaches

Group communication

Sender not aware of identities of receivers

Can be implemented as overlay network or over IP multicast to manage group membership, detect failures, provide reliability, provide message ordering, etc.

Applications of group communication:

Financial events dissemination to group of clients

Collaborative applications such as multiuser games

Managing replicated data

Systems monitoring and management, such as load balancing

Group communication

Programming model:

- A *group* of entities is key concept
- Managing *group membership* via *join* and *leave*

Types of communication based on number of receivers:

- *Unicast*: one receiver entity
- *Multicast*: a group of receivers (group communication)
- *Broadcast*: all entities in the distributed systems are receivers

Advantages of multicast over multiple unicast operations:

- Performance can be optimized when transmitting message to group of entities
- Can be more fault-tolerant

Group communication

Process group:

- A group consists of *processes*
- Supports message delivery to all processes in group
- Basic communication support no marshalling/unmarshalling

Object group:

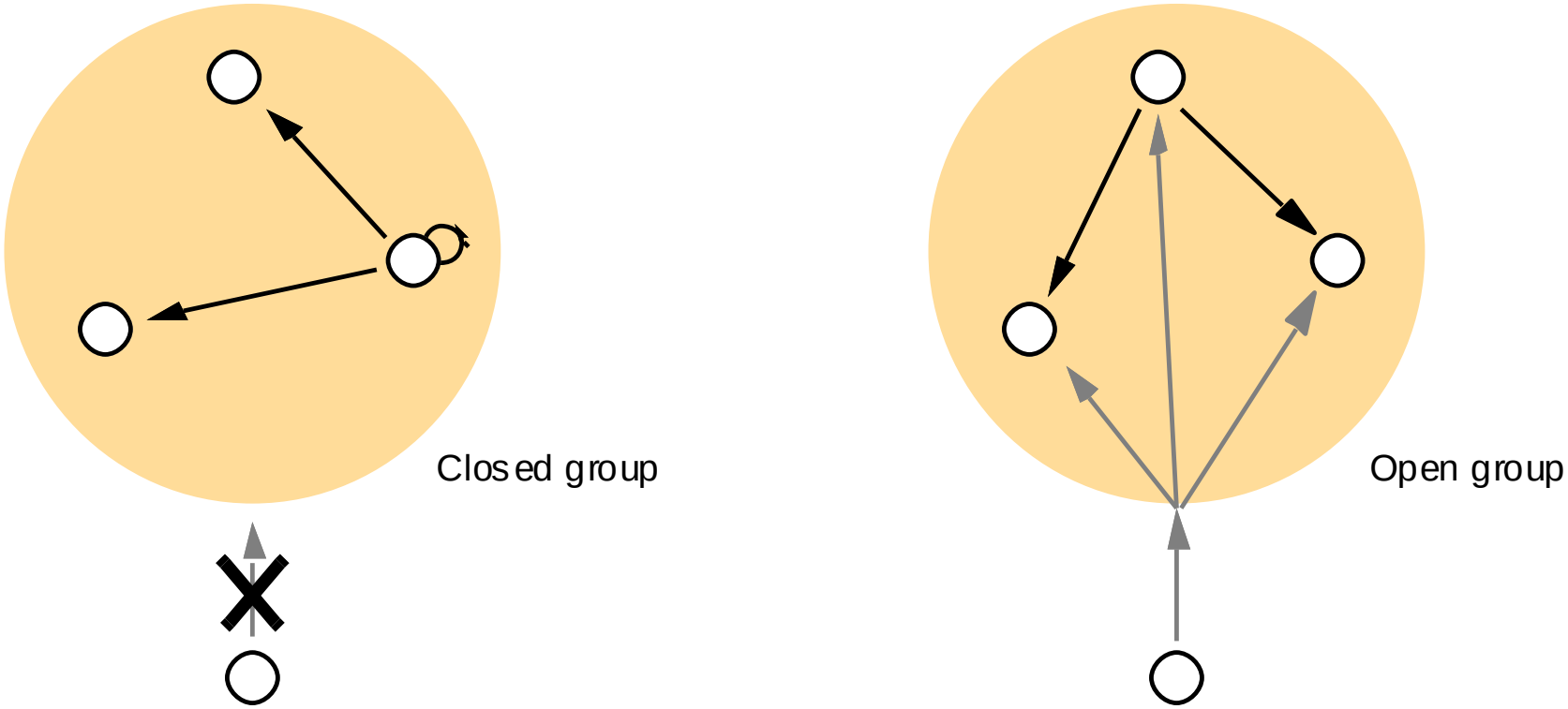
- A group consists of *objects*
- Can invoke methods concurrently in a group of replicated objects
- Can provide marshalling/unmarshalling
- Can hide replication from sending client by having a local proxy representing the object group

Group communication

Closed versus *open* groups:

- Closed group: only a group member can multicast to group (including itself)
- Open group: process outside group my multicast to group
- Mailing lists can be closed groups in some cases and open groups in others, depending on how the mailing list is utilized

Figure 6.2
Open and closed groups



Group communication

Overlapping versus *non-overlapping* groups:

- Overlapping: process can be a member in multiple groups
- Non-overlapping: process can be member of at most one group

Synchronous versus *asynchronous* groups:

Multicast algorithm depends on the group communication properties

Group communication

Reliability in multicast:

- Three properties: Integrity, Validity, Agreement
- Integrity: correctly deliver a message at most once
- Validity: a message that is sent will eventually be delivered
- Agreement: message delivered to one process in the group will be delivered to all group members

Ordered multicast:

- FIFO ordering: messages received in same order they were sent
- Causal ordering: if a message *happens before* another message, this
- Total ordering: messages that are delivered in same order in all processes

Group communication

Group membership management services:

- Interface needed to provide: creating/destroying groups; adding/withdrawing a process from a group
- Failure detection: detect failed/unreachable processes and remove from group
- Membership notifications: notify members of processes added/withdrawn/removed
- Group address expansion: service expands group identifier into group member addresses (dynamically)

Figure 6.3
The role of group membership management

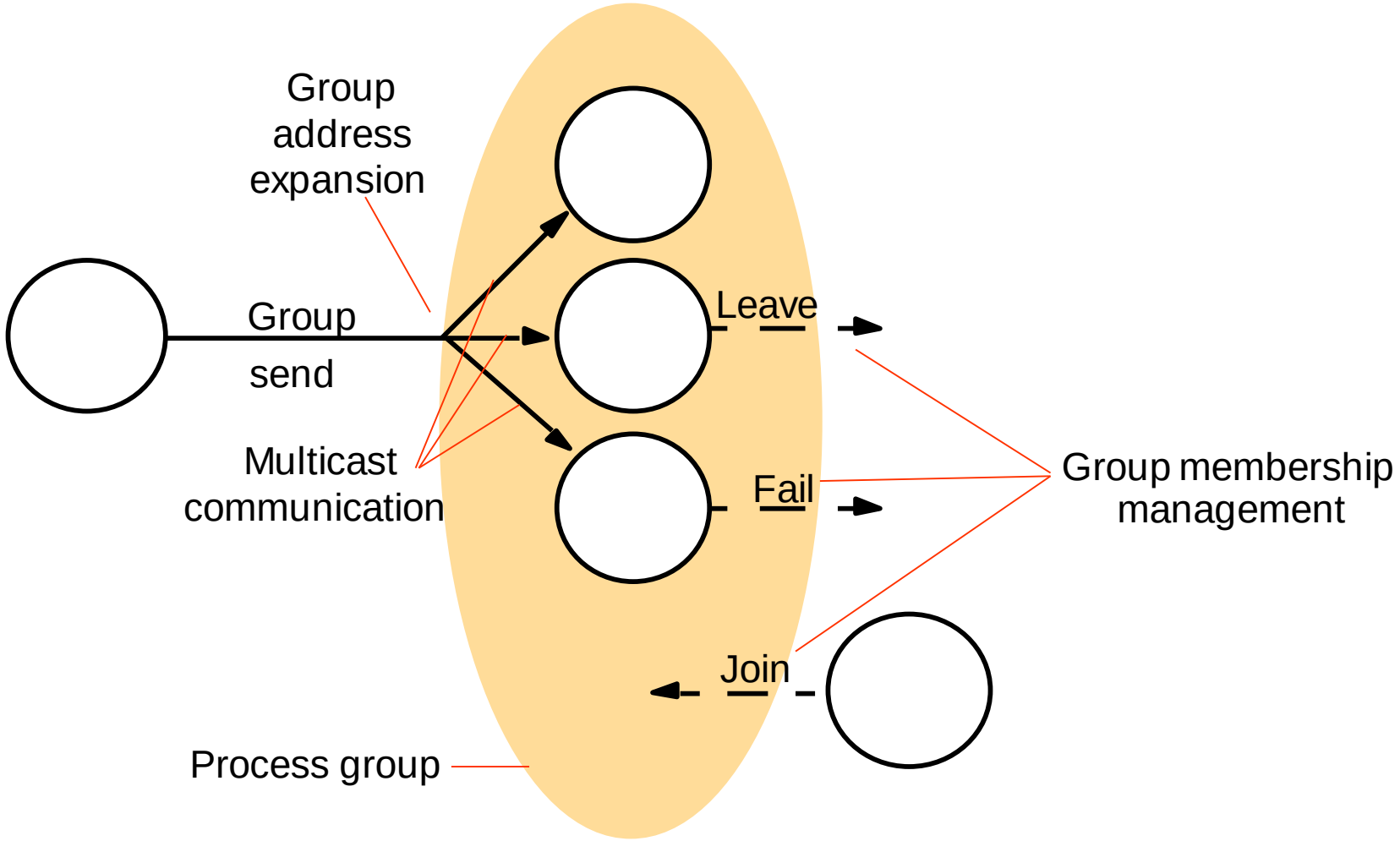


Figure 6.4
The architecture of JGroups

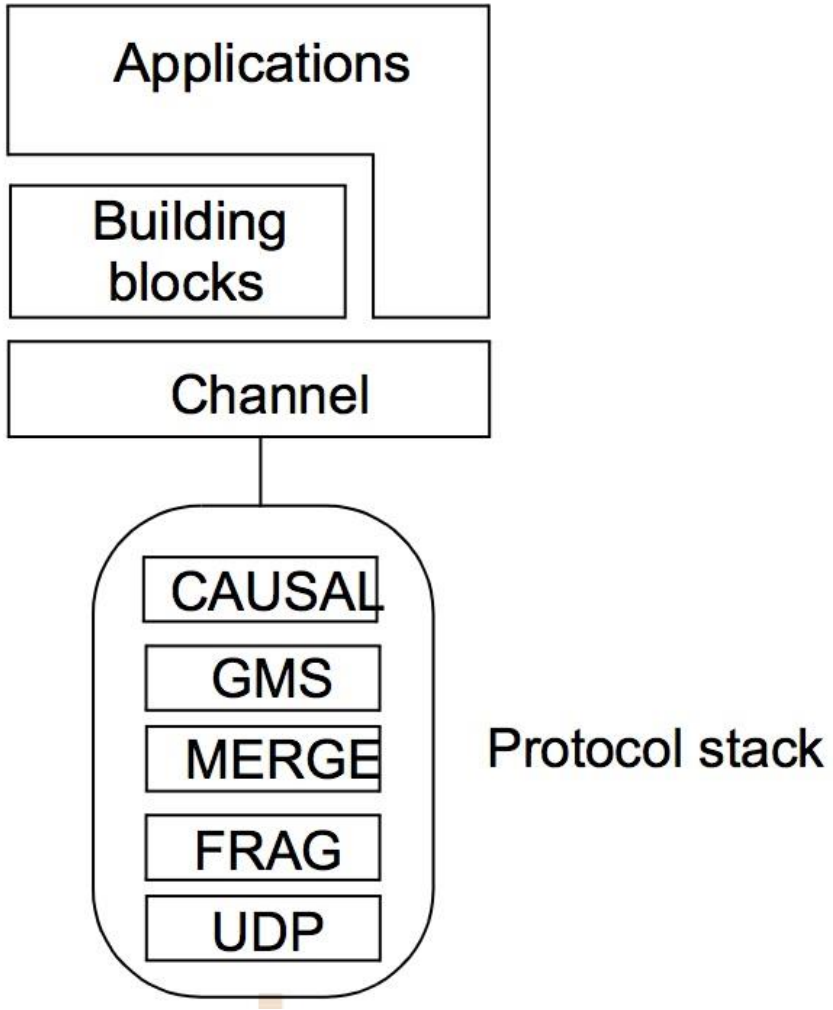


Figure 6.5
Java class *FireAlarmJG*

```
import org.jgroups.JChannel;
public class FireAlarmJG {
public void raise() {
    try {
        JChannel channel = new JChannel();
        channel.connect("AlarmChannel");
        Message msg = new Message(null, null, "Fire!");
        channel.send(msg);
    }
    catch(Exception e) {
    }
}
```

Figure 6.6
Java class *FireAlarmConsumerJG*

```
import org.jgroups.JChannel;

    public class FireAlarmConsumerJG {
        public String await() {
        try {
                JChannel channel = new JChannel();
                channel.connect("AlarmChannel");
                Message msg = (Message) channel.receive(0);
                return (String) msg.GetObject();
        } catch(Exception e) {
                return null;
        }
    }
}
```

Overview of Chapter

Introduction to Indirect Communication

Group communication

Publish-subscribe systems

Message queues

Shared memory approaches

Publish-subscribe systems

Also known as *distributed event-based systems*

Most widely used type of indirect communication

Definitions:

Publishers: publish *structured events*

Subscribers: express interest in particular published events (*subscriptions*); these are arbitrary patterns over the structured events

Publish-subscribe system: matches subscriptions against published events by delivering *event notifications*

Publish-subscribe systems

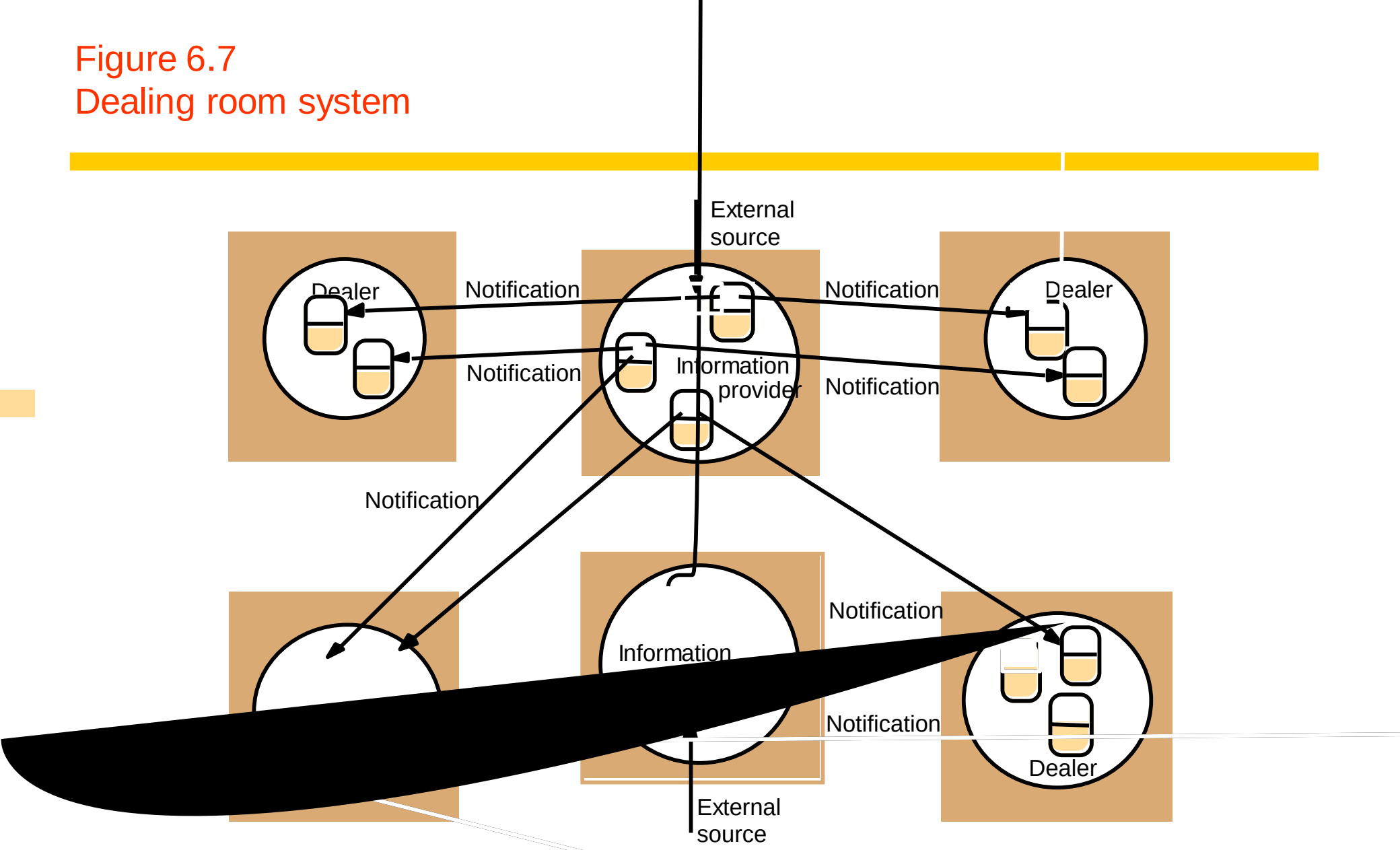
Applications:

- Financial systems
- Live feeds including RSS (Rich Site Summary or RDF Site Summary) feeds
- Cooperative works that involves informing participants of shared events
- Ubiquitous computing support
- Monitoring applications, such as network monitoring

Example:

- *Dealing room system*

Figure 6.7
Dealing room system



Publish-subscribe systems

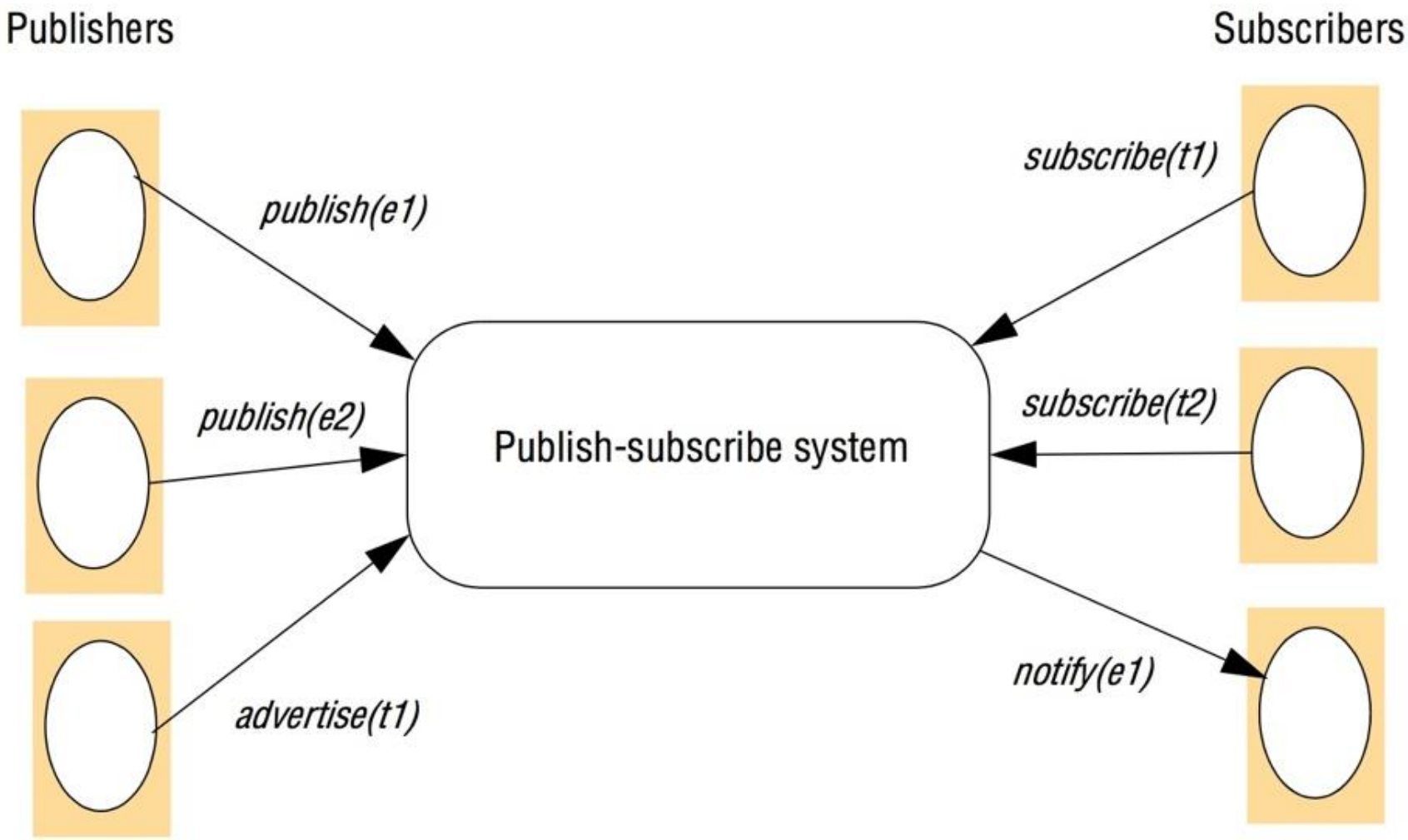
Characteristics:

- Heterogeneity: systems can be heterogeneous
- Asynchronicity: no synchronization between publishers and subscribers
- Variety of delivery guarantees: depends on application

Programming model:

- Small set of operations
- *publish(e)* to publish an event by a publisher
- *subscribe(f)* to subscribe to a particular pattern specified by a *filter f*
- *notify(e)* event arrives at subscriber
- *unsubscribe(f)* to revoke a subscription
- *advertise(f)* – declare nature of future events
- *unadvertise(f)* revoke an advertisement

Figure 6.8
The publish-subscribe paradigm



Publish-subscribe systems

Subscription (filter) models:

- Channel-based: events published to named channels, subscribers subscribe to channels to get all events on that channel (no filtering)
- Topic- or subject-based: published events specify topic; subscribers specify subscription topics
- Content-based: events are structured; filters specify subscriptions as conditions on attributes in events (queries)
- Type-based: events are objects; subscription filters specify types of events

Other models:

- Object of interest (Jini)
- Context-based (e.g. location in mobile systems)

Publish-subscribe systems

Implementation issues:

- Can be quite complex
- *Broker:* node that maintains subscriptions for subscribers and receives notifications from publishers; matches event notications to subscriptions (filters)
- Centralized versus distributed: centralized would have a single node known as *event broker* to handle all operations; distributed uses network of broker nodes
- Peer-to-peer: no distinction between broker, subscriber and publisher nodes

Figure 6.9
A network of brokers

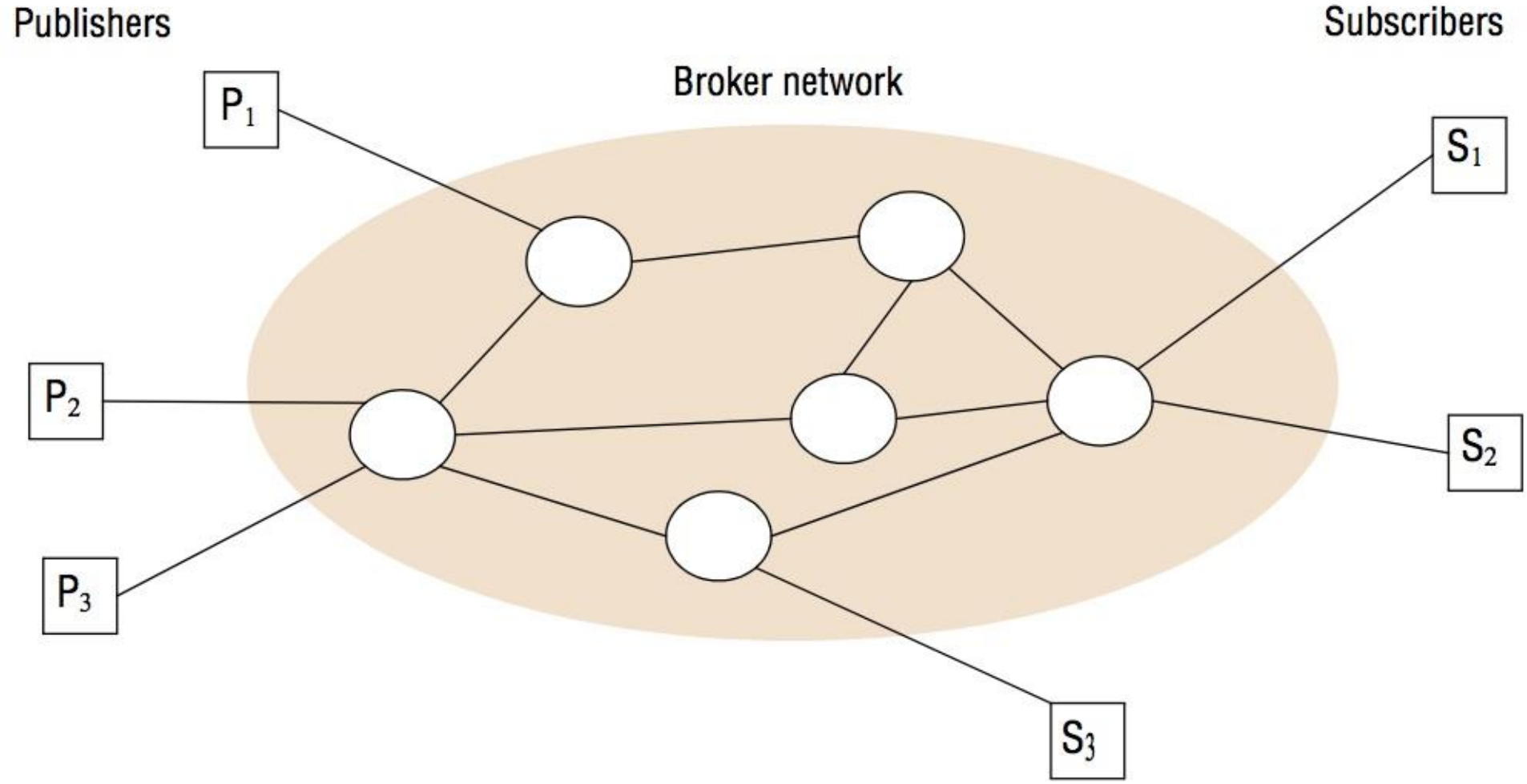
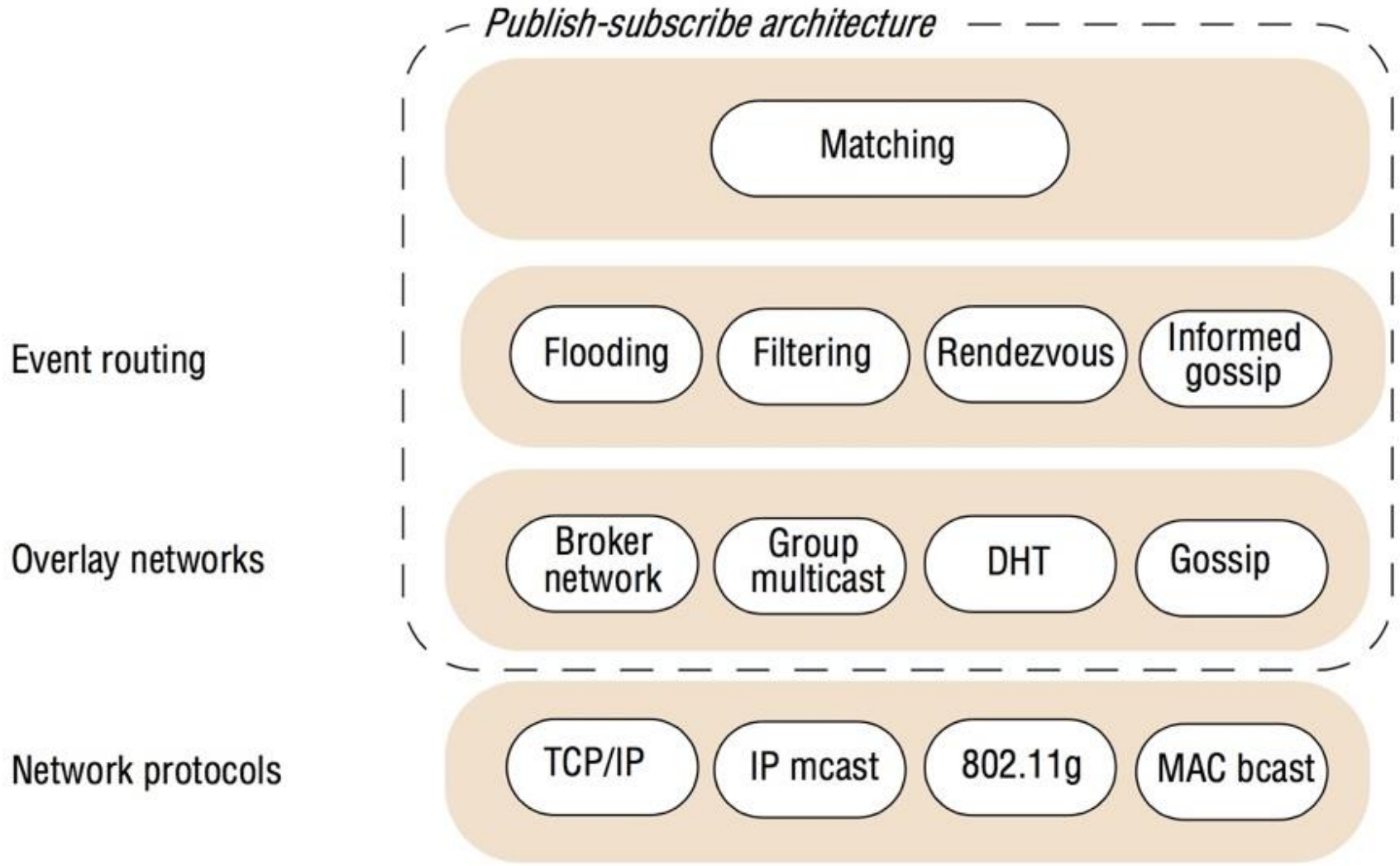


Figure 6.10
The architecture of publish-subscribe systems



Publish-subscribe systems

Content-based routing:

- Flooding: Published event notifications sent to all subscriber nodes, matching done at receiving node; alternatively, subscriptions sent to publishers for matching then publisher uses point-to-point communication
- Filtering*: broker nodes* process event notifications and subscriptions, do matching (*filtering-based routing)*
- Advertisements: propagate advertisements towards subscribers
- Rendezvous: partition event notifications among broker (rendezvous) nodes (*rendezvous-based routing*) can use DST (distributed hash tables)

Figure 6.11
Filtering-based routing

```
upon receive publish(event e) from node x                    1
    matchlist := match(e, subscriptions)                     2
    send notify(e) to matchlist;                             3
    fwdlist := match(e, routing);   4
    send publish(e) to fwdlist - x;  5
upon receive subscribe(subscription s) from node x           6
    if x is client then     7
        add x to subscriptions;    8
    else add(x, s) to routing; 9
    send subscribe(s) to neighbours - x;        10
```

Figure 6.12
Rendezvous-based routing

```
upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist :=match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;
upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;
```

Figure 6.13
Example publish-subscribe system

System (and further reading)	*Subscription model*	*Distribution model*	*Event routing*
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvouz [Oki *et al.* 1993]	Topic-based	Distributed	Ffiltering
Scribe [Castro *et al.* 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni *et al.* 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga *et al.* 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	

Overview of Chapter

Introduction to Indirect Communication

Group communication

Publish-subscribe systems

Message queues

Shared memory approaches

Message queues

Also known as distributed message queues or Message Oriented Middleware (MOM)

Achieve point-to-point communication (one sender, one receiver) that is time and space uncoupled

Several commercial implementations to achieve EAI (Enterprise Application Integration):

Also used TPS (Transaction Processing Systems)

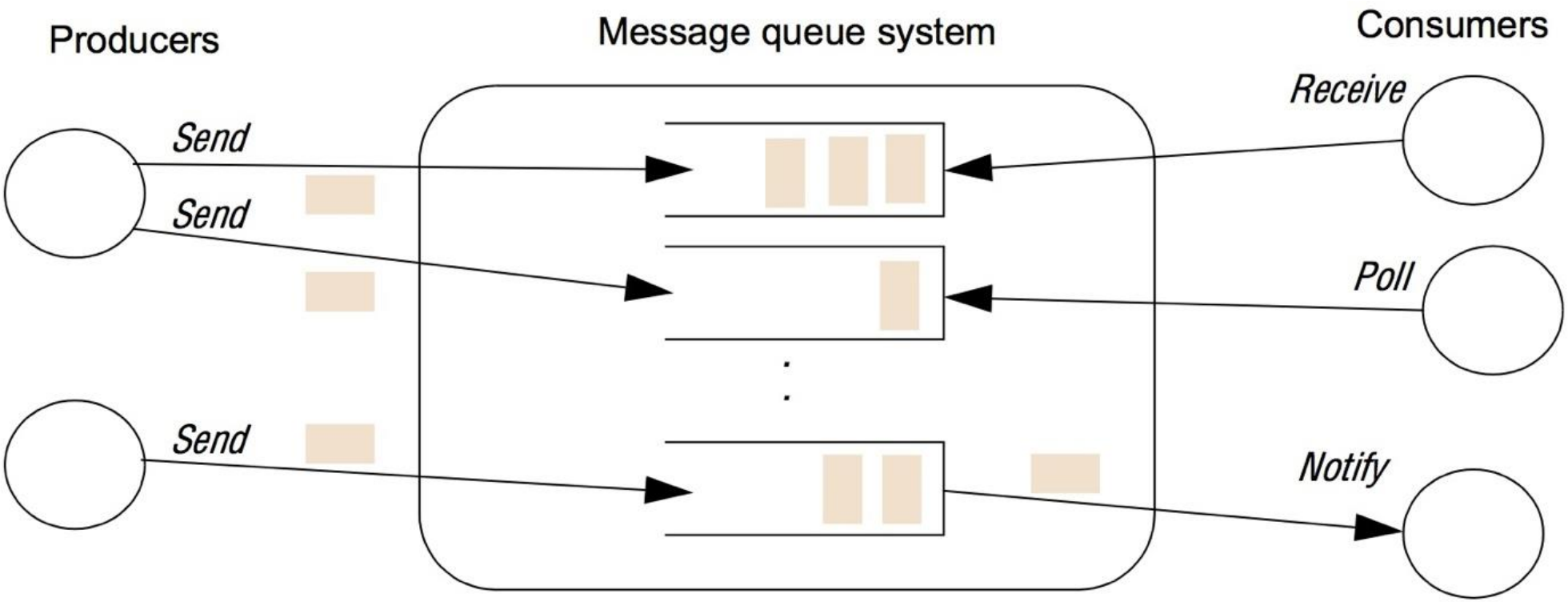
Message queues

Programming model:

- System designers creates queues each queue has a unique identifier
- *Producer* sends message to specific queue
- *Consumer* receives messages from this queue
- Three styles of receive: blocking (process waits for message), non-blocking (polling process checks queue), notify (event notification when message in queue)
- Queueing policy is usually FIFO but can also use message priority in some systems

Also used TPS (Transaction Processing Systems)

Figure 6.14
The message queue paradigm



Message queues

Programming model (cont.):

- Message *destination* is queue id
- *Attributes (metadata)* associated with a message may include: priority, delivery mode, etc.
- Messages are *opaque* (serialized string of bytes, often large)
- Messages can be selected based on *metedata*
- In Oracle AQ, messages can be relational database rows (records), and can be queried using SQL query language

- Messages are *persistent* stored until they are consumed (delivered)
- Hence, delivery is *reliable* guaranteed one-time delivery of each message

Message queues

Other features:

- Messages can be contained in a *transaction* (see Chapter 16)
  transaction services can be provided by middleware
- Messages may be *transformed* on delivery for example, *format* transformation to deal with heterogeneity between sender and receiver system
- Support for *security* may be provided (e.g. SSL (secure socket layer))

Note: somewhat similar to message-passing systems (Chapter 4) except queues are *explicit* (in MPI queues are *implicit* buffers at receiver)

Message queues

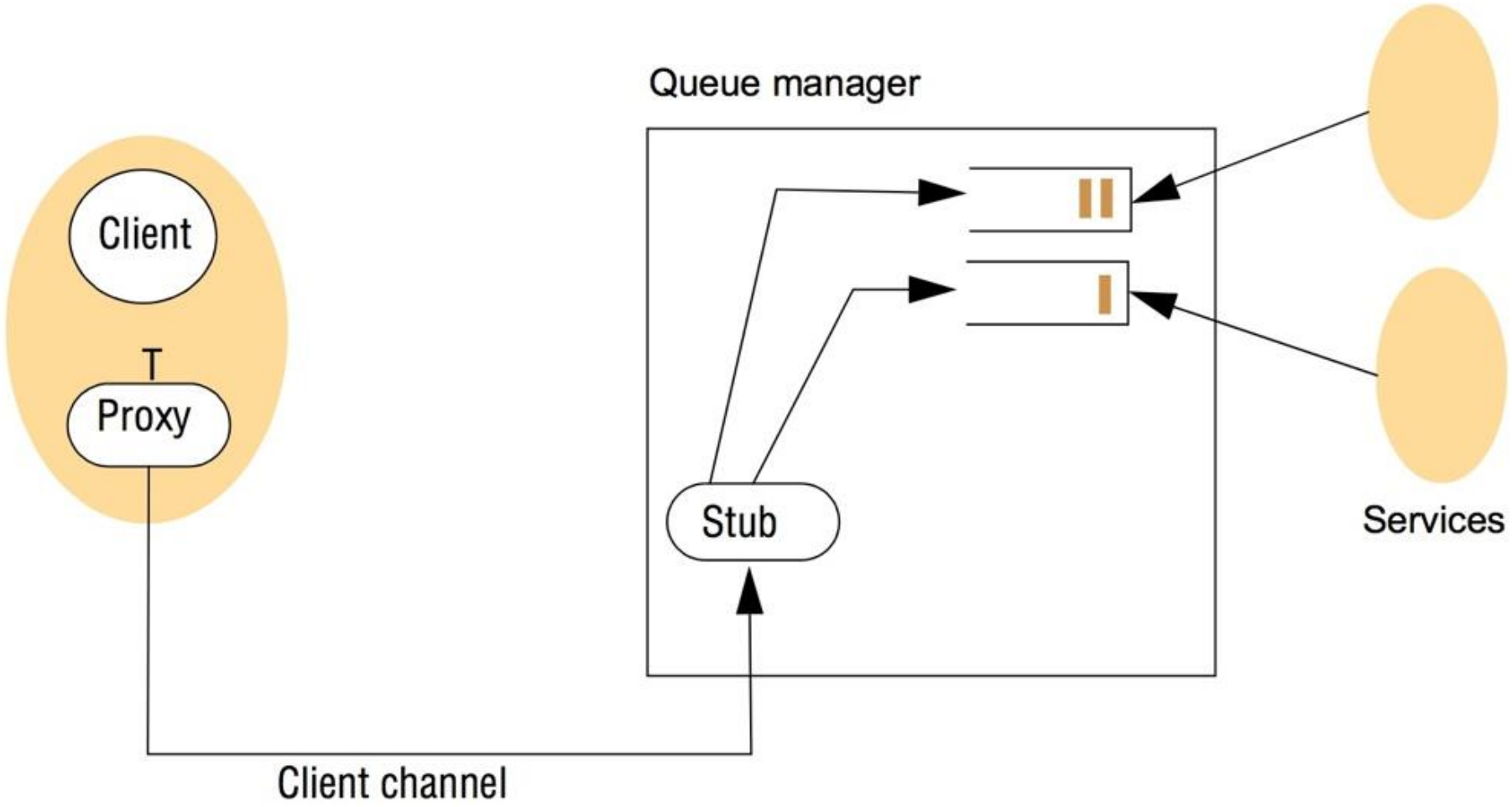
Implementation issues:

- Queueing system could be *centralized* or *distributed*
- *Centralized* node can be performance bottleneck and subject to failure, biut simpler to implement
- *Distributed* nodes (possibly with replication) is more complex but more reliable

Example

- *Queue managers* manage queues through *MQI* (Message Queueing Interface)
- MQI operations include MQCONN (connect), MQDISC (disconnect), MQPUT (send), MQGET (receive)
- Clients have *proxy* to set up a *client channel* to queue manager

Figure 6.15
A simple networked topology in WebSphere MQ



Message queues

Example MQ (cont.):

System can have multiple queue manager nodes if distributed

Message channel can be set up between two queue managers to forward messages

System topology is set up during design

Another example: JMS (*Java Messaging Service)*

JMS client can be *producer* or *consumer*

Figure 6.16
The programming model offered by JMS

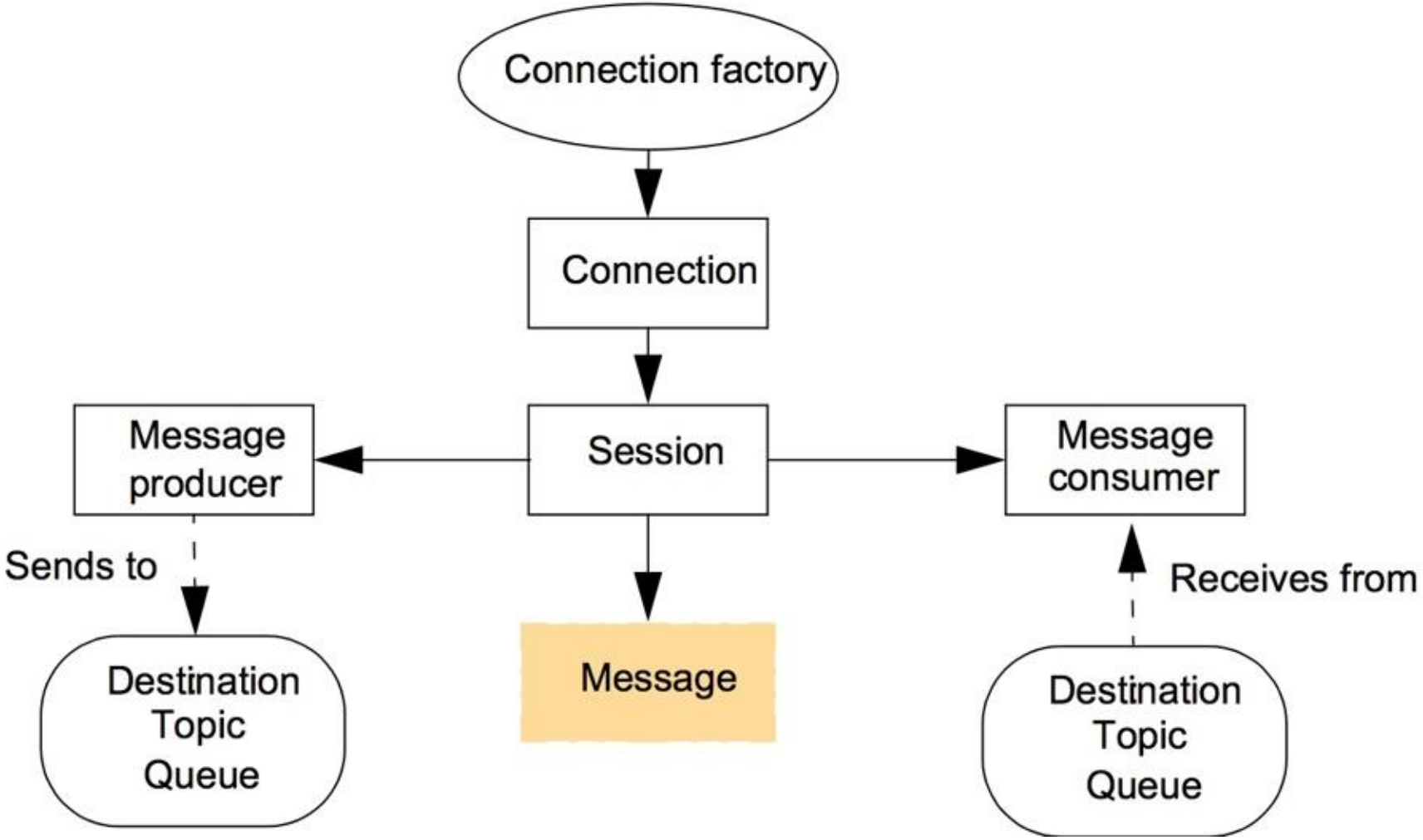


Figure 6.17
Java class *FireAlarmJMS*

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {

public void raise() {
      try {                                                    1
            Context ctx = new InitialContext();          2
            TopicConnectionFactory  topicFactory  =     3
            (TopicConnectionFactory)ctx.lookup  ("TopicConnectionFactory");  4
            Topic topic = (Topic)ctx.lookup("Alarms");   5
            TopicConnection  topicConn =     6
                  topicConnectionFactory.createTopicConnection();  7
            TopicSession topicSess = topicConn.createTopicSession(false, 8
                  Session.AUTO_ACKNOWLEDGE);                                  9
            TopicPublisher  topicPub  = topicSess.createPublisher(topic);          10;
            TextMessage msg = topicSess.createTextMessage();          11
             msg.setText("Fire!"); 12
            topicPub.publish(message);     13
            } catch  (Exception  e) {      14
            } 15
}
```

Figure 6.18
Java class *FireAlarmConsumerJMS*

```
import javax.jms.*;  import javax.naming.*;
public class FireAlarmConsumerJMS
public String await(){
    try {                                                    1
        Context ctx = new InitialContext();      2
        TopicConnectionFactory topicFactory =   3
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory"); 4
          Topic topic = (Topic)ctx.lookup("Alarms"); 5
          TopicConnection topicConn =          6
                topicConnectionFactory.createTopicConnection();        7
          TopicSession topicSess = topicConn.createTopicSession(false,        8
                  Session.AUTO_ACKNOWLEDGE);        9
          TopicSubscriber topicSub = topicSess.createSubscriber(topic);            10
          topicSub.start();  11
          TextMessage msg = (TextMessage) topicSub.receive();         12
          return msg.getText(); 13
          } catch (Exception e) {     14
                      return null;  15
    }16
}
```

Overview of Chapter

Introduction to Indirect Communication

Group communication

Publish-subscribe systems

Message queues

Shared memory approaches

Shared Memory Approaches

Various approaches

Distributed shared memory

Tuple space communication

Distributed Shared Memory (DSM)

Abstraction for sharing data between nodes (computers) that do not have a physical shared memory

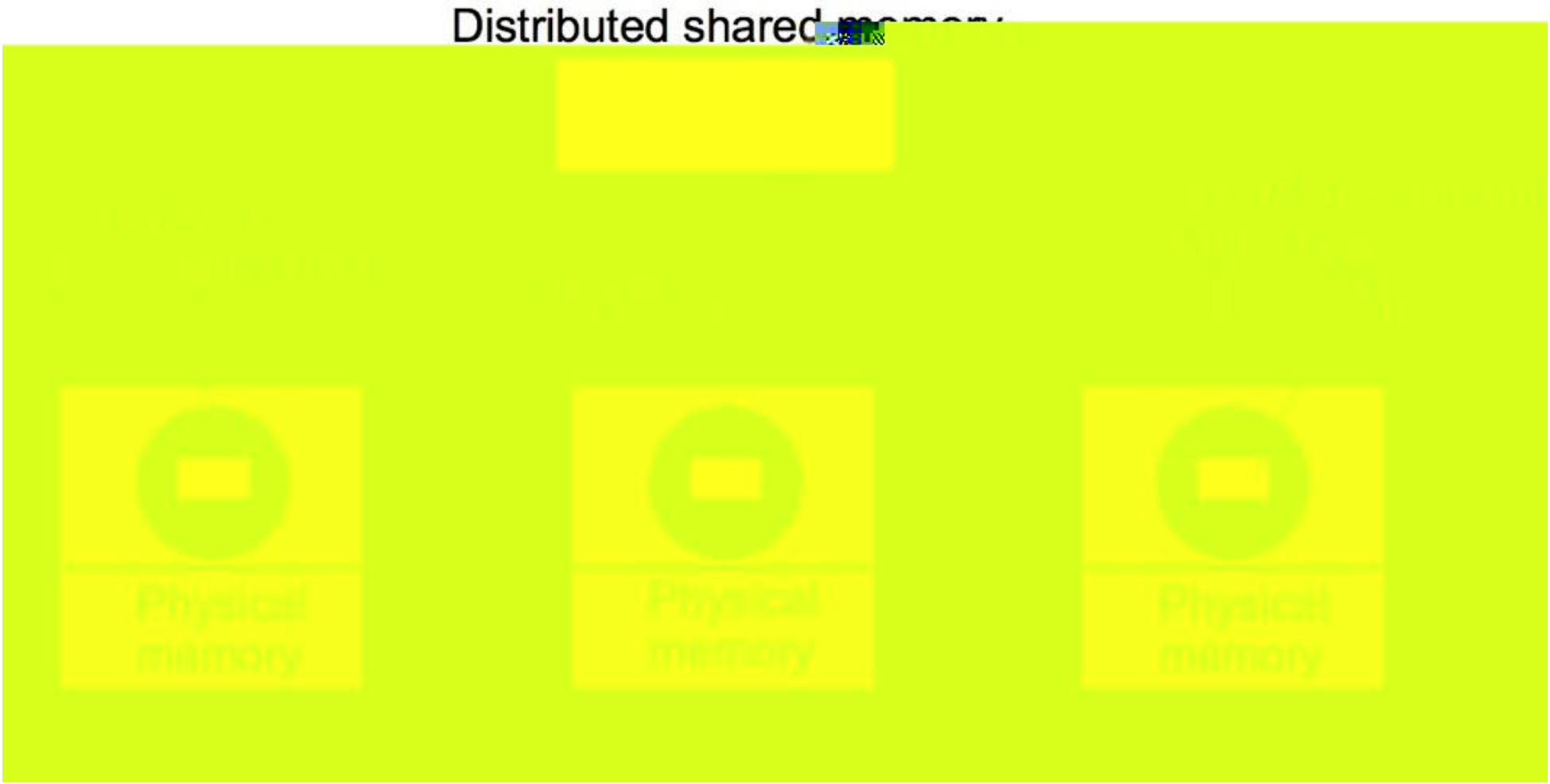
Operations include *read* and *update*

Updates can be seen by all nodes

Avoids message passing

DSM typically replicated on each node items can be efficiently accessed (replication issues discussed in Chapter 18)

Figure 6.19
The distributed shared memory abstraction



Tuple space communication

Programming model:

- Tuples are stored in a shared tuple space
- Operations include: *read; write* (create a new tuple); and *take* (read and delete a tuple)
- Reads and takes are specified by pattern matching queries
- Tuples are *immutable* (cannot be changed)
- Properties include both *time uncoupling* and *space uncoupling*

Example systems:

Linda and JavaSpaces

Will not cover in detail

Figure 6.20
The tuple space abstraction

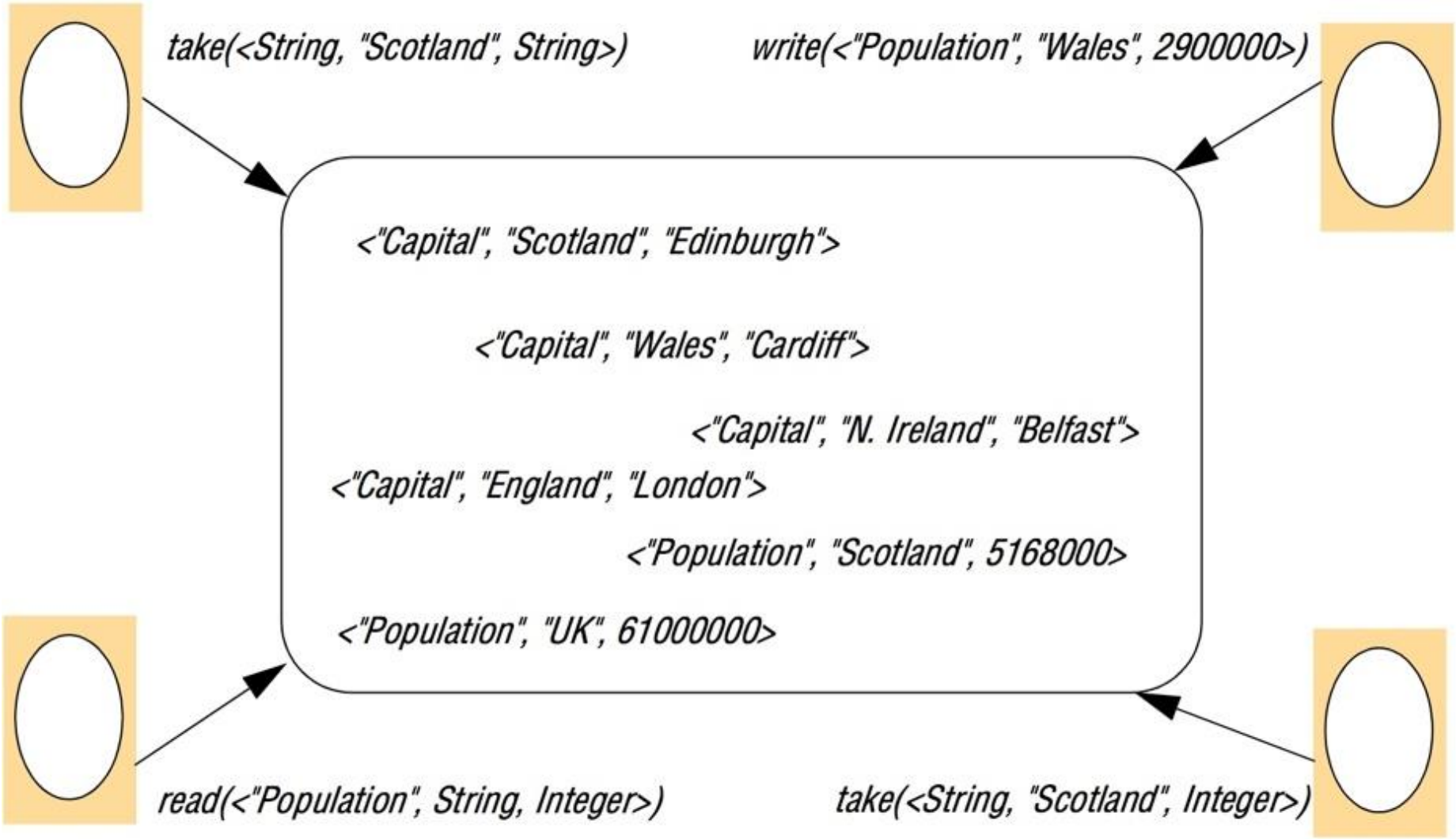


Figure 6.21 (1)
Replication and the tuple space operations [Xu and Liskov 1989]

write

1. The requesting site multicasts the *write* request to all members of the view;
2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

read

1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others); 4 Step 1 is repeated until at least one response is received.

continued on next slide

Figure 6.21 (continued)
Replication and the tuple space operations [Xu and Liskov 1989]

take *Phase 1: Selecting the tuple to be removed*

1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple

1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

Figure 6.22
Partitioning in the York Linda Kernel

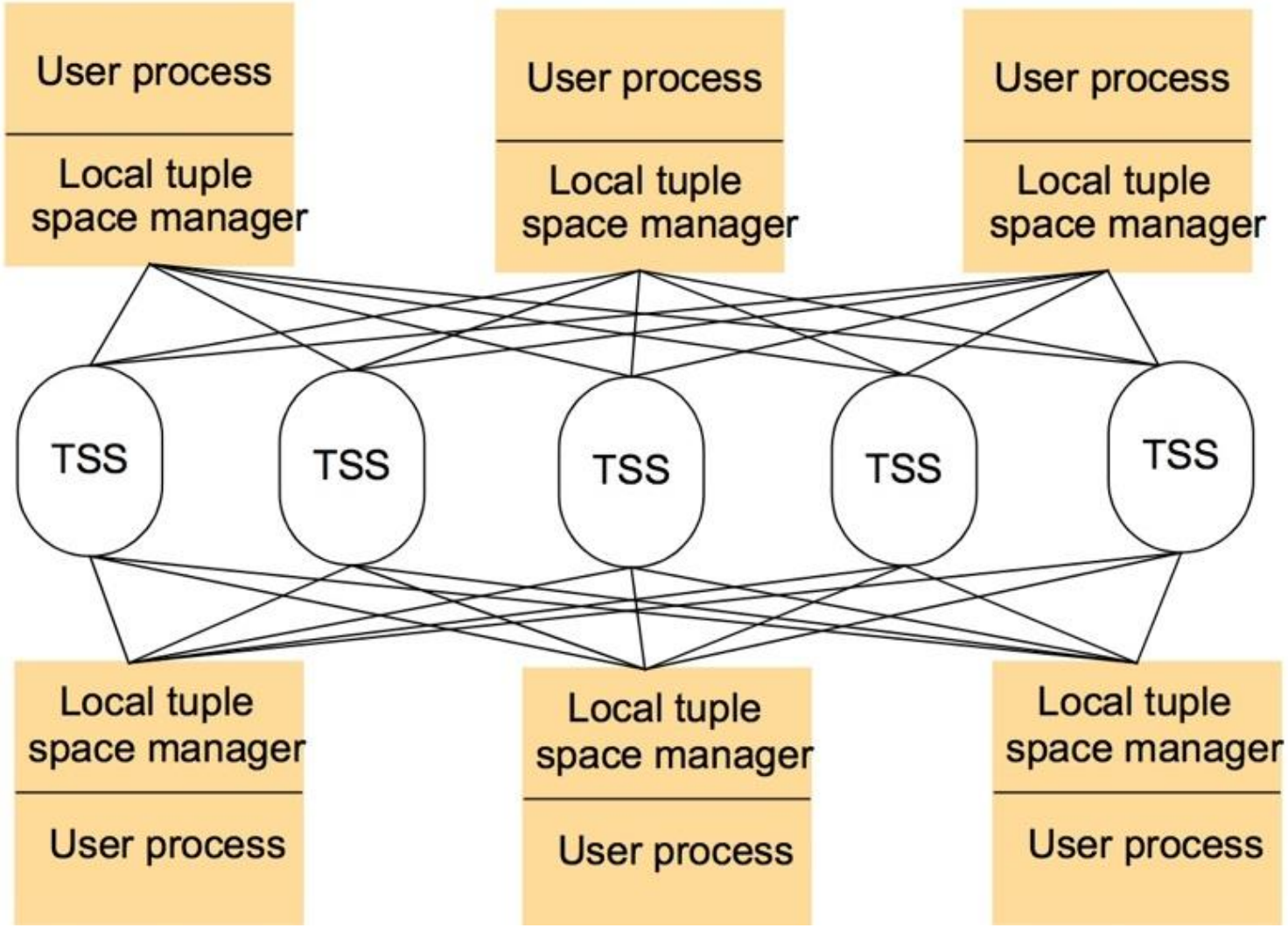


Figure 6.23
The JavaSpaces API

Operation	*Effect*
Lease write(Entry e, Transaction txn, long lease)	Places an entry into a particular JavaSpace
Entry read(Entry tmpl, Transaction txn, long timeout)	Returns a copy of an entry matching a specified template
Entry readIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
Entry take(Entry tmpl, Transaction txn, long timeout)	Retrieves (and removes) an entry matching a specified template
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)	Notifies a process if a tuple matching a specified template is written to a JavaSpace

Figure 6.24
Java class *AlarmTupleJS*

```
import net.jini.core.entry.*;
public class AlarmTupleJS implements Entry {
    public String alarmType;
        public AlarmTupleJS() { }
    }
    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;}
    }
}
```

Figure 6.25
Java class *FireAlarmJS*

```
import net.jini.space.JavaSpace;
public class FireAlarmJS {
public void raise() {
      try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
       catch (Exception e) {
      }
    }
}
```

Figure 16.26
Java class *FireAlarmReceiverJS*

```
import net.jini.space.JavaSpace;
public class FireAlarmConsumerJS {
public String await(){
    try {
            JavaSpace space = SpaceAccessor.findSpace();
            AlarmTupleJS template = new AlarmTupleJS("Fire!");
            AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                    Long.MAX_VALUE);
            return recvd.alarmType;
    }
        catch (Exception e) {
            return null;
          }
      }
}
```

Figure 6.27
Summary of indirect communication styles

	Groups	*Publish-subscribe systems*	*Message queues*	*DSM*	*Tuple spaces*
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
Main intent	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based publish-subscribe only	No	No	Yes