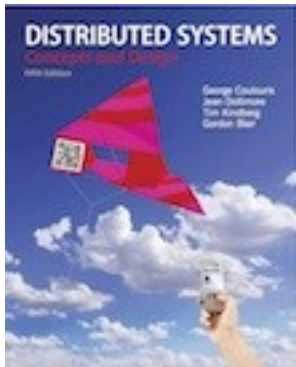


Slides for Chapter 6: Indirect Communication



From **Coulouris, Dollimore, Kindberg and Blair**
**Distributed Systems:
Concepts and Design**

Edition 5, © Addison-Wesley 2012

Figure 6.1
Space and time coupling in distributed systems

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

Figure 6.2

Open and closed groups

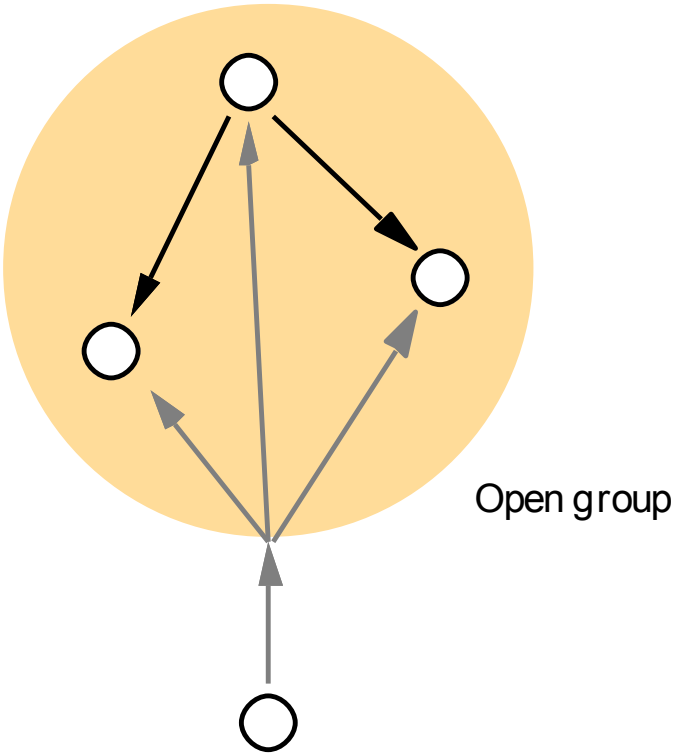
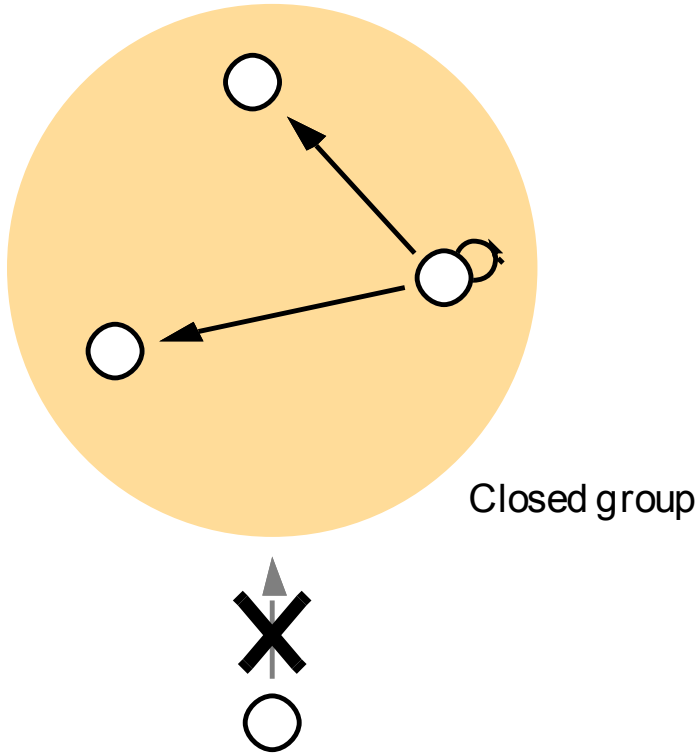


Figure 6.3 The role of group membership management

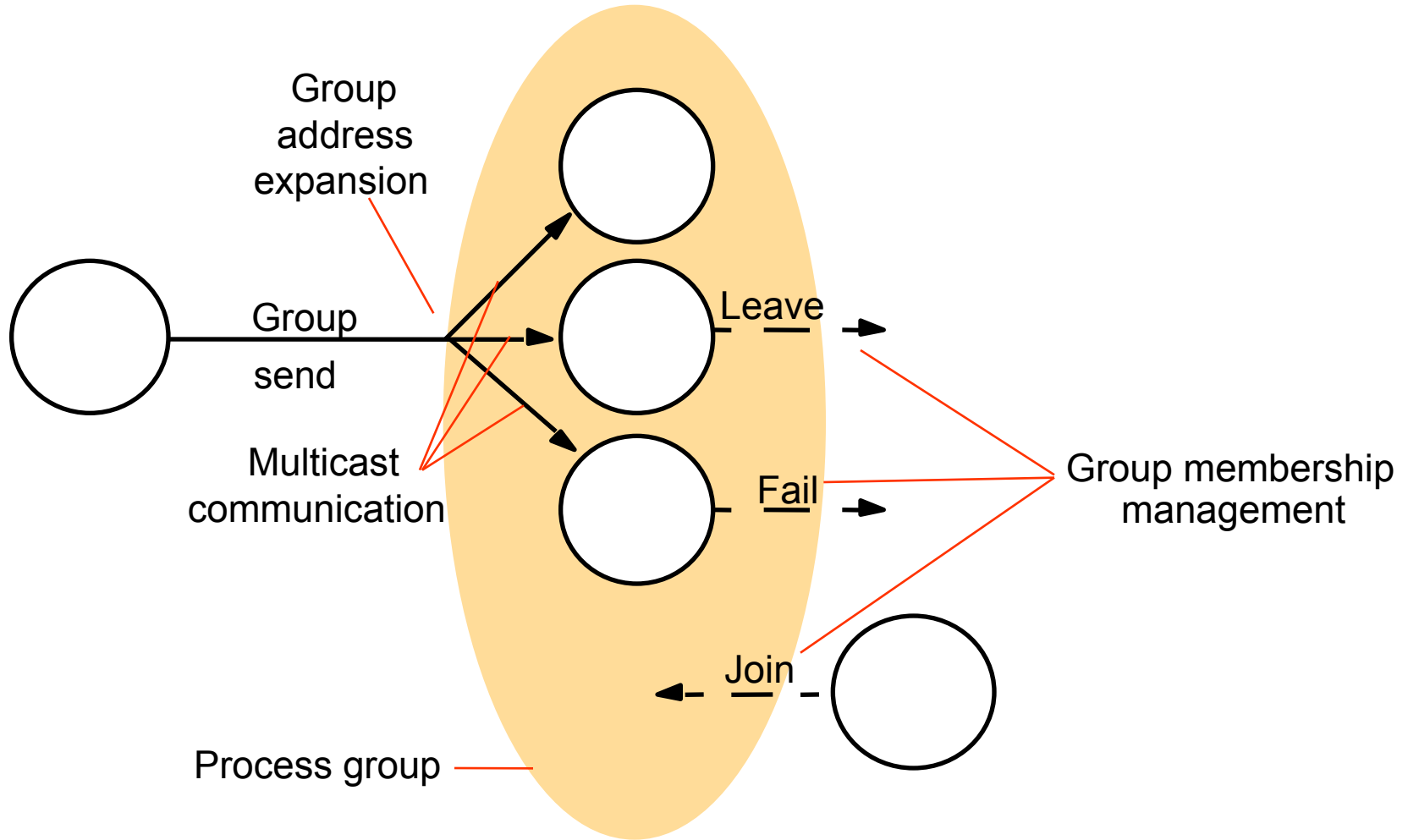


Figure 6.4 The architecture of JGroups

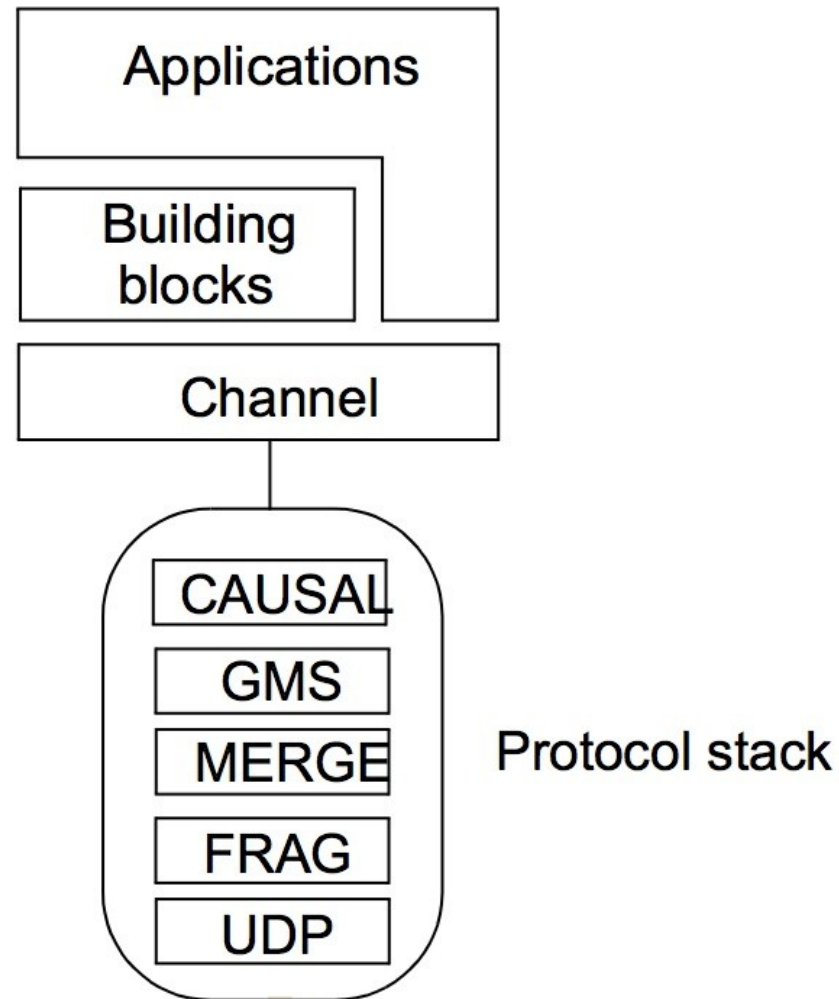


Figure 6.5

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;
        }
    }
}
```

Figure 6.7
Dealing room system

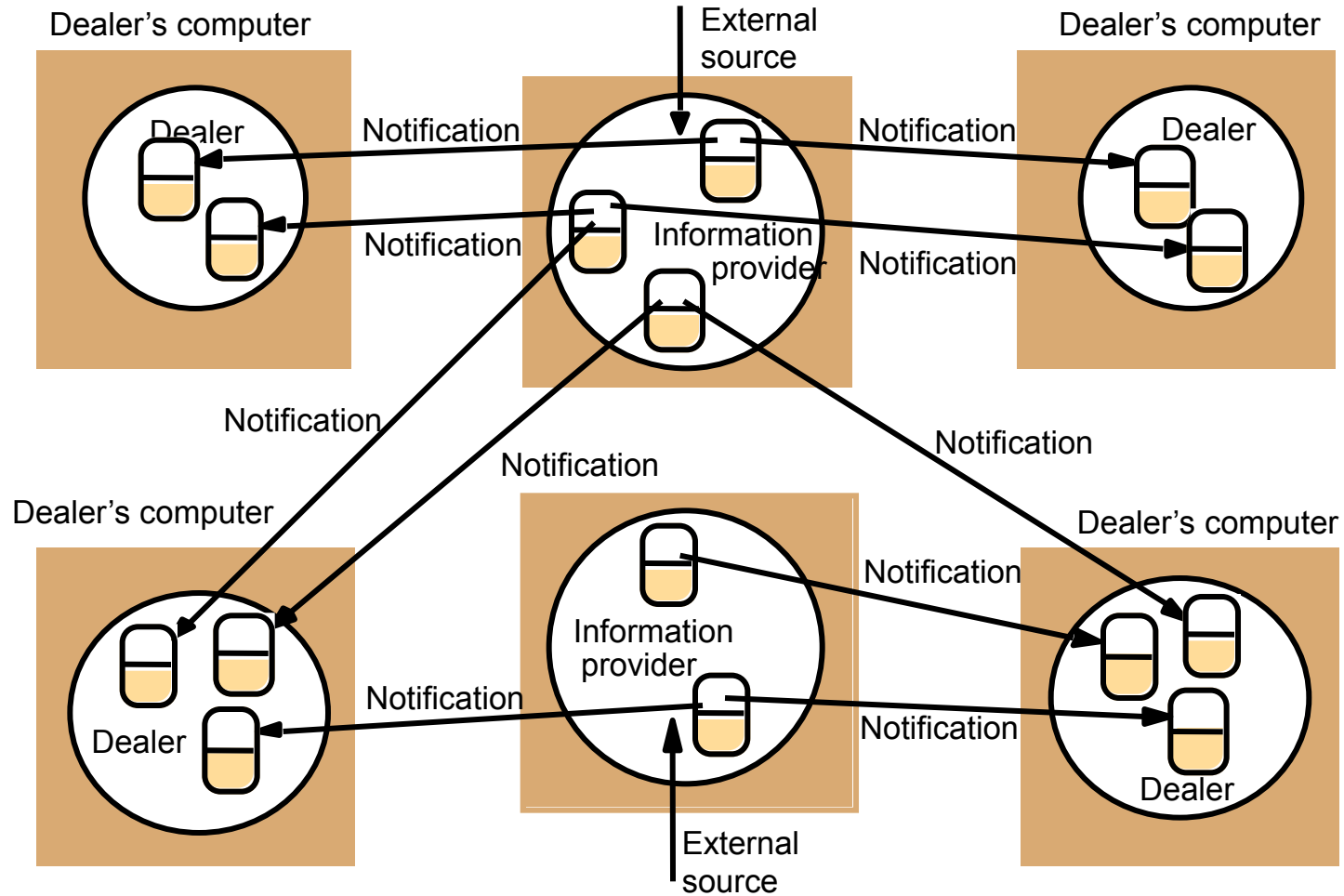


Figure 6.8
The publish-subscribe paradigm

Publishers

Subscribers

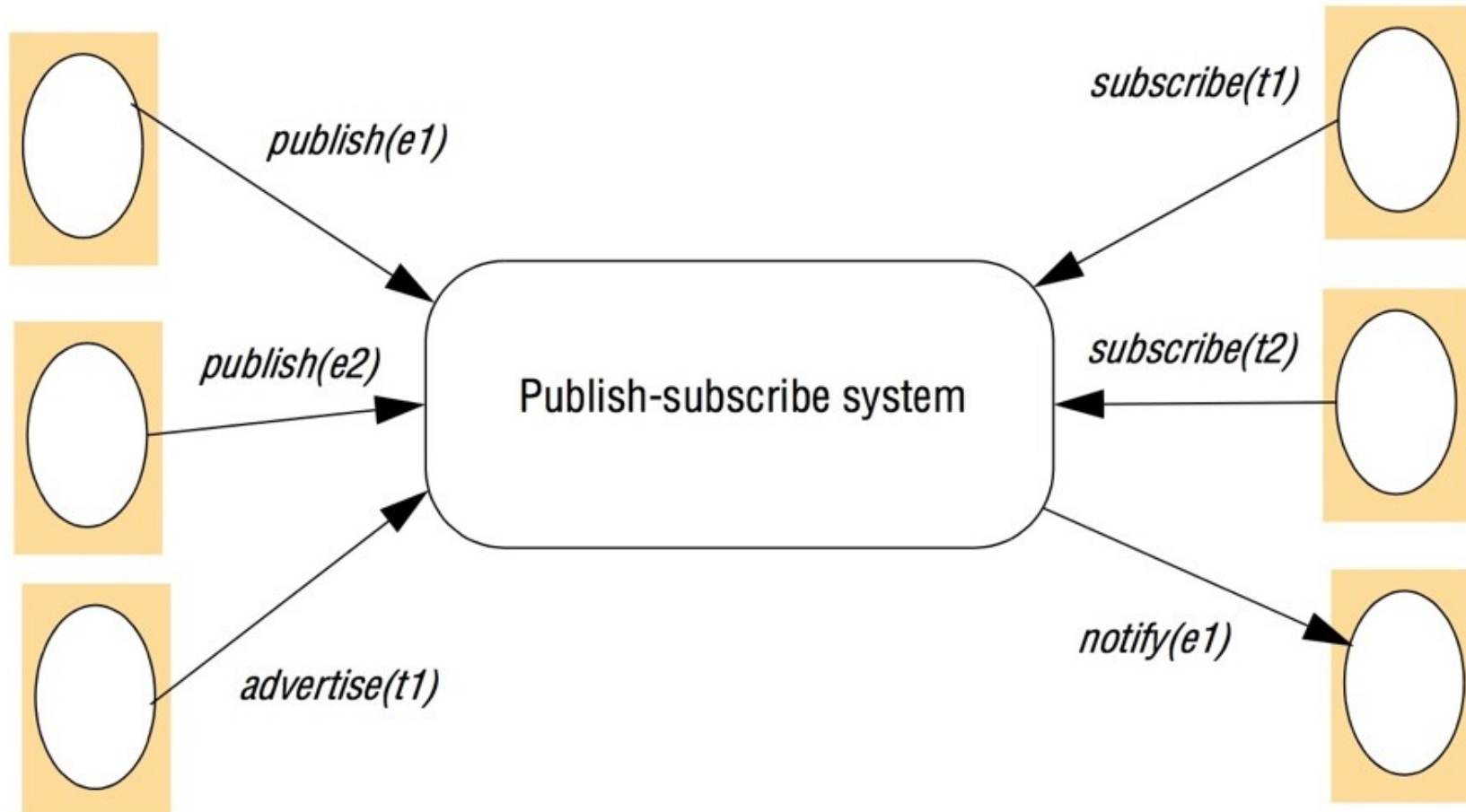


Figure 6.9
A network of brokers

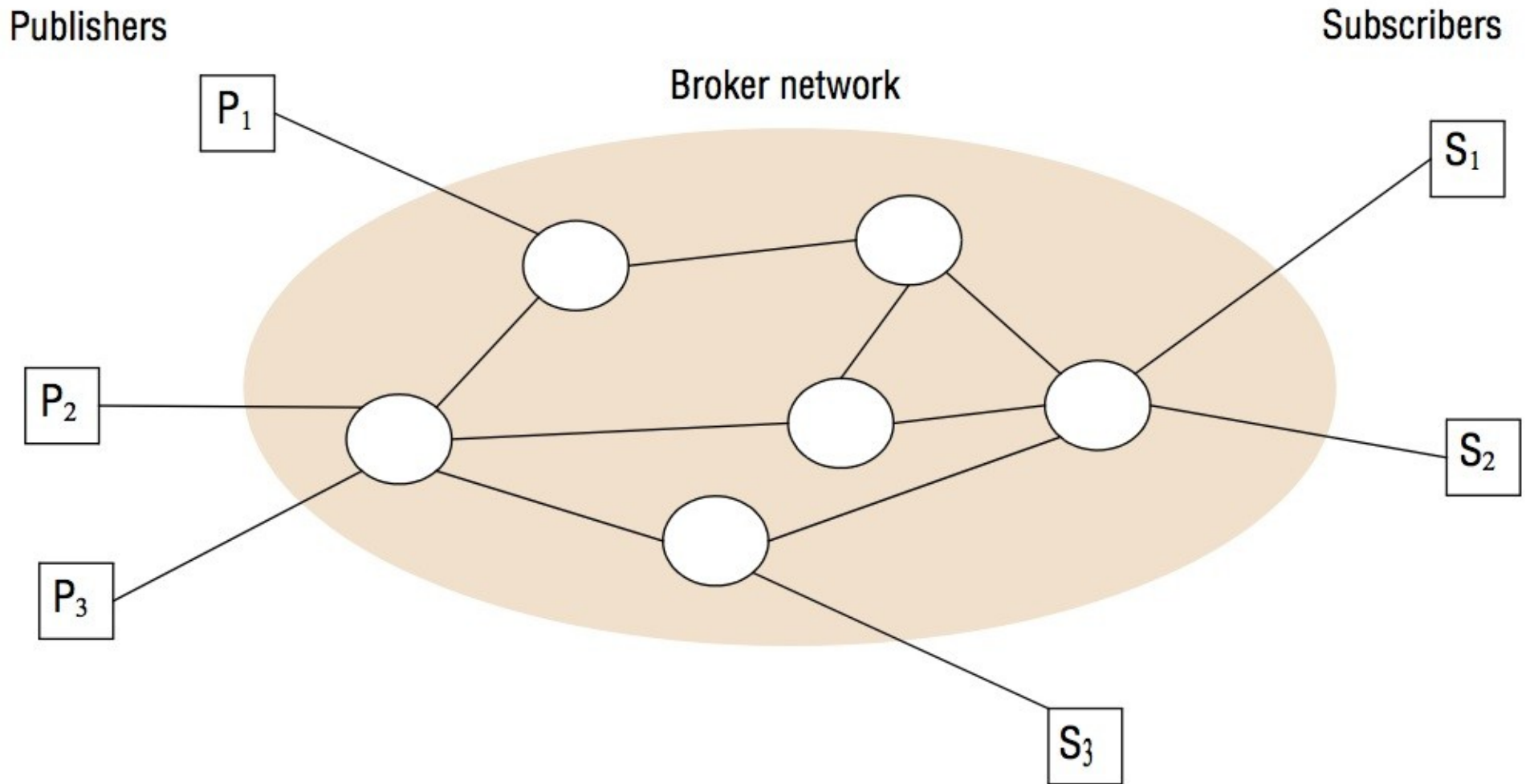


Figure 6.10

The architecture of publish-subscribe systems

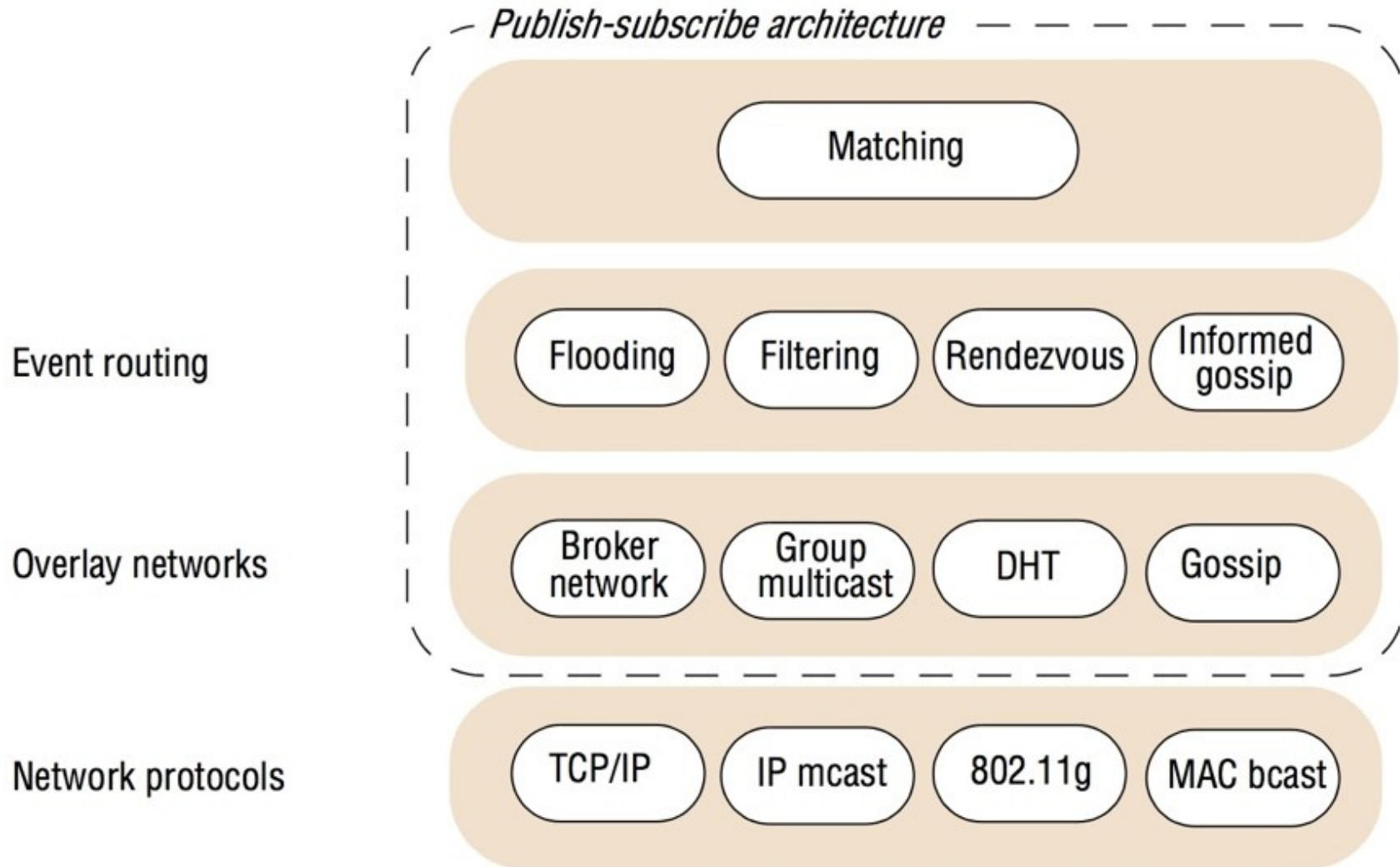


Figure 6.11 Filtering-based routing

```
upon receive publish(event e) from node x 1  
  matchlist := match(e, subscriptions) 2  
  send notify(e) to matchlist; 3  
  fwddlist := match(e, routing); 4  
  send publish(e) to fwddlist - 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

<i>System (and further reading)</i>	<i>Subscription model</i>	<i>Distribution model</i>	<i>Event routing</i>
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvous [Oki <i>et al.</i> 1993]	Topic-based	Distributed	Filtering
Scribe [Castro <i>et al.</i> 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni <i>et al.</i> 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga <i>et al.</i> 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	Flooding
Meghdoot [Gupta <i>et al.</i> 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni <i>et al.</i> 2005]	Content-based	Peer-to-peer	Informed gossip

Figure 6.14
The message queue paradigm

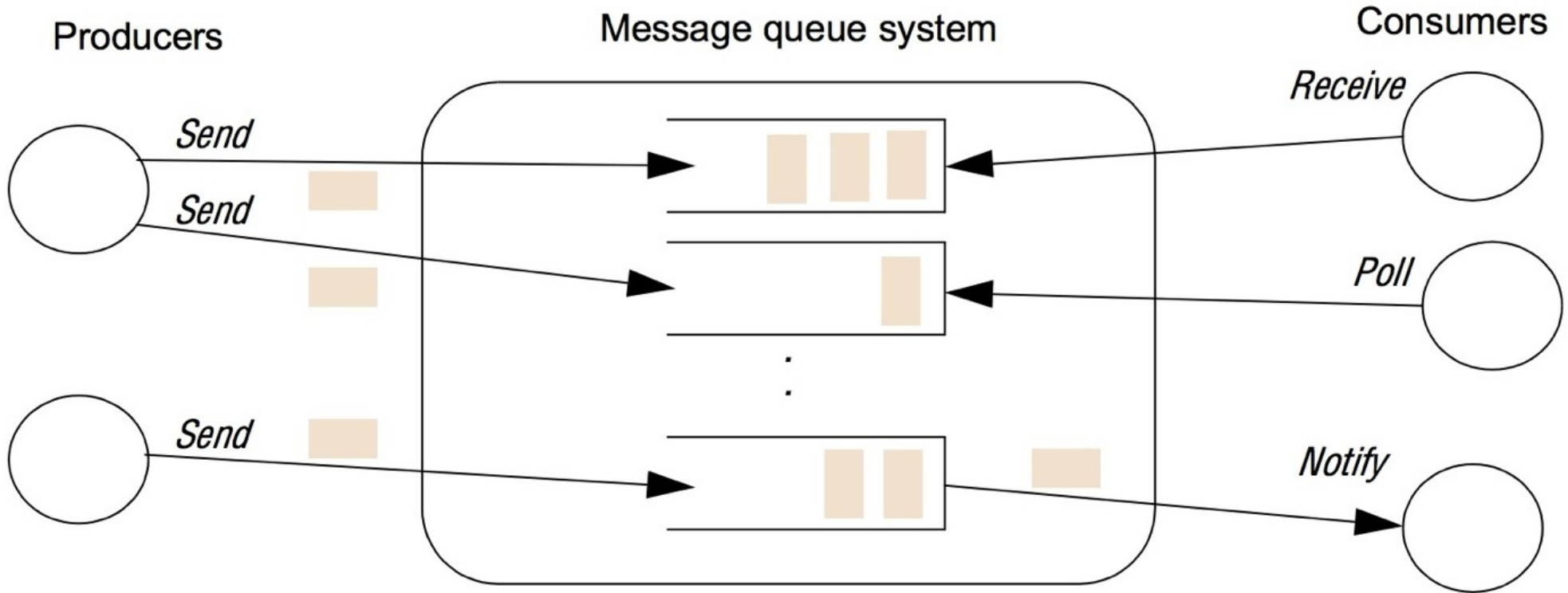


Figure 6.15
A simple networked topology in WebSphere MQ

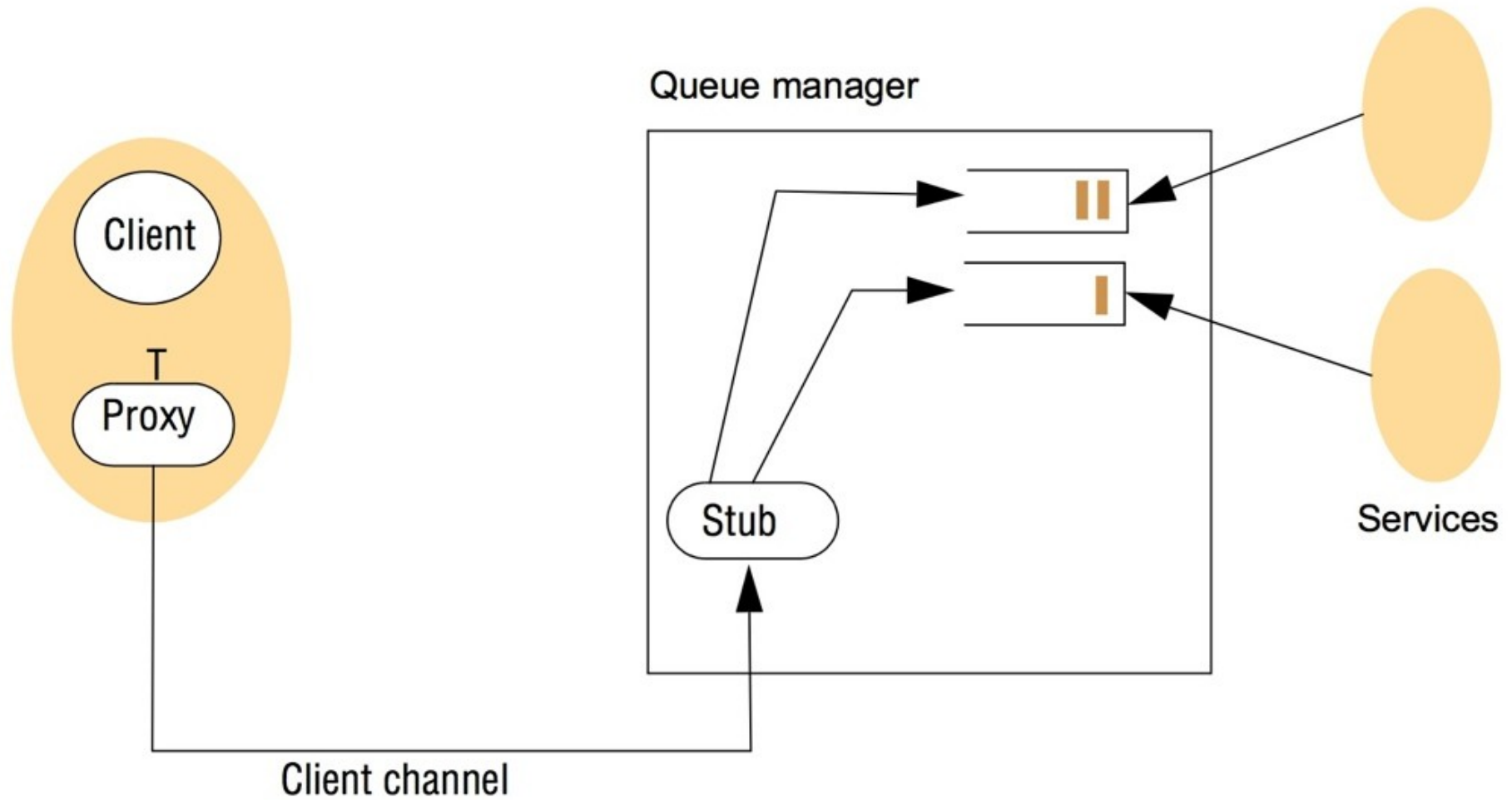


Figure 6.16
The programming model offered by JMS

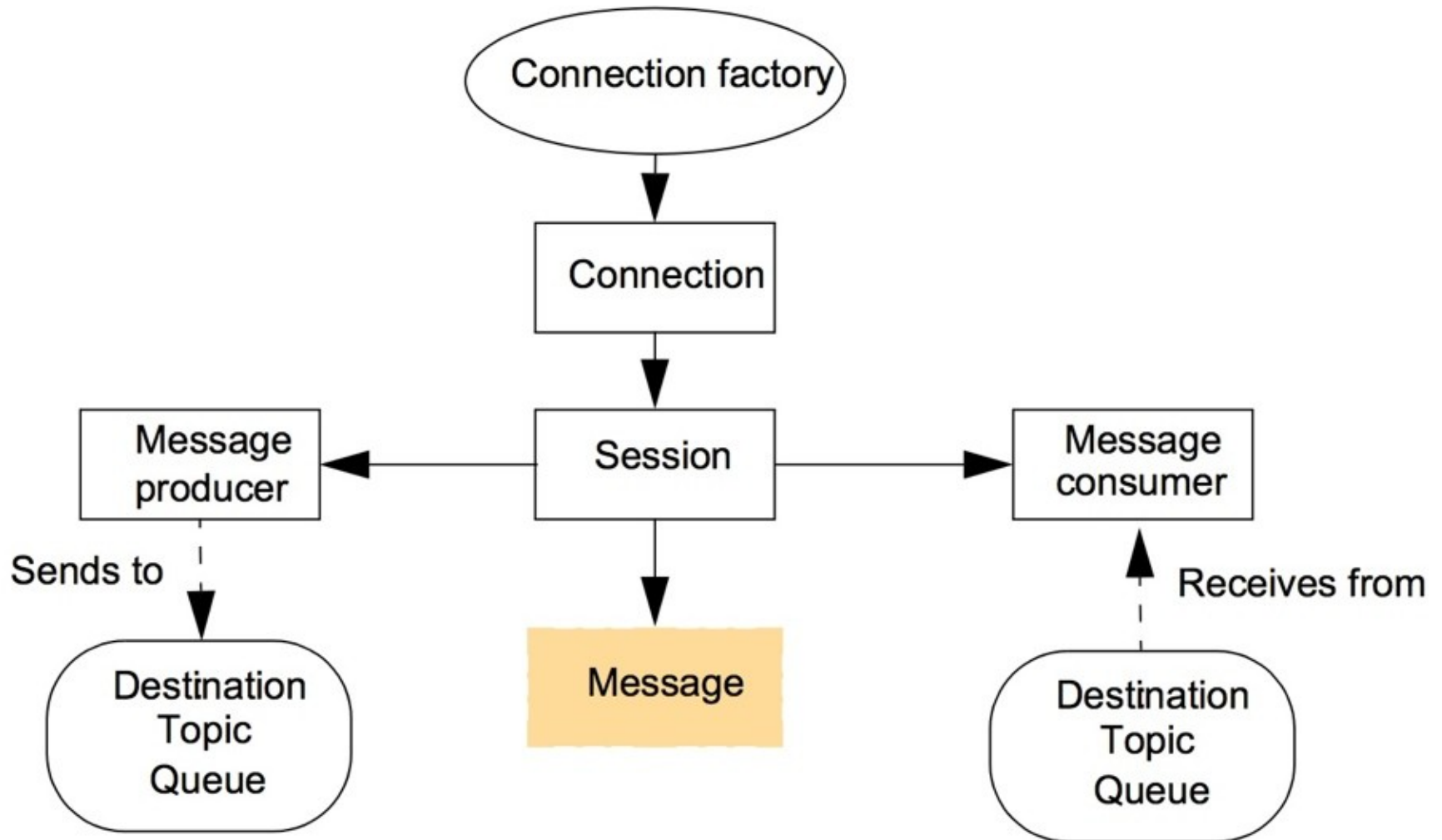


Figure 6.17

Java class *FireAlarmJMS*

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

    public void raise() {
        try {
            Context ctx = new InitialContext();
            TopicConnectionFactory topicFactory =
                (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
            Topic topic = (Topic)ctx.lookup("Alarms");
            TopicConnection topicConn =
                topicConnectionFactory.createTopicConnection();
            TopicSession topicSess = topicConn.createTopicSession(
                false, Session.AUTO_ACKNOWLEDGE);
            TopicPublisher topicPub = topicSess.createPublisher(topic);
            TextMessage msg = topicSess.createTextMessage();
            msg.setText("Fire!");
            topicPub.publish(message);
        } catch (Exception e) {
        }
    }
}
```

Figure 6.18

Java class *FireAlarmConsumerJMS*

```
import javax.jms.*; import javax.naming.*;
public class FireAlarmConsumerJMS
public String await() {
    try {
        Context ctx = new InitialContext();
        TopicConnectionFactory topicFactory =
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
        Topic topic = (Topic)ctx.lookup("Alarms");
        TopicConnection topicConn =
            topicConnectionFactory.createTopicConnection();
        TopicSession topicSess = topicConn.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
        TopicSubscriber topicSub = topicSess.createSubscriber(topic);
        topicSub.start();
        TextMessage msg = (TextMessage) topicSub.receive();
        return msg.getText();
    } catch (Exception e) {
        return null;
    }
}
```

Figure 6.19
The distributed shared memory abstraction

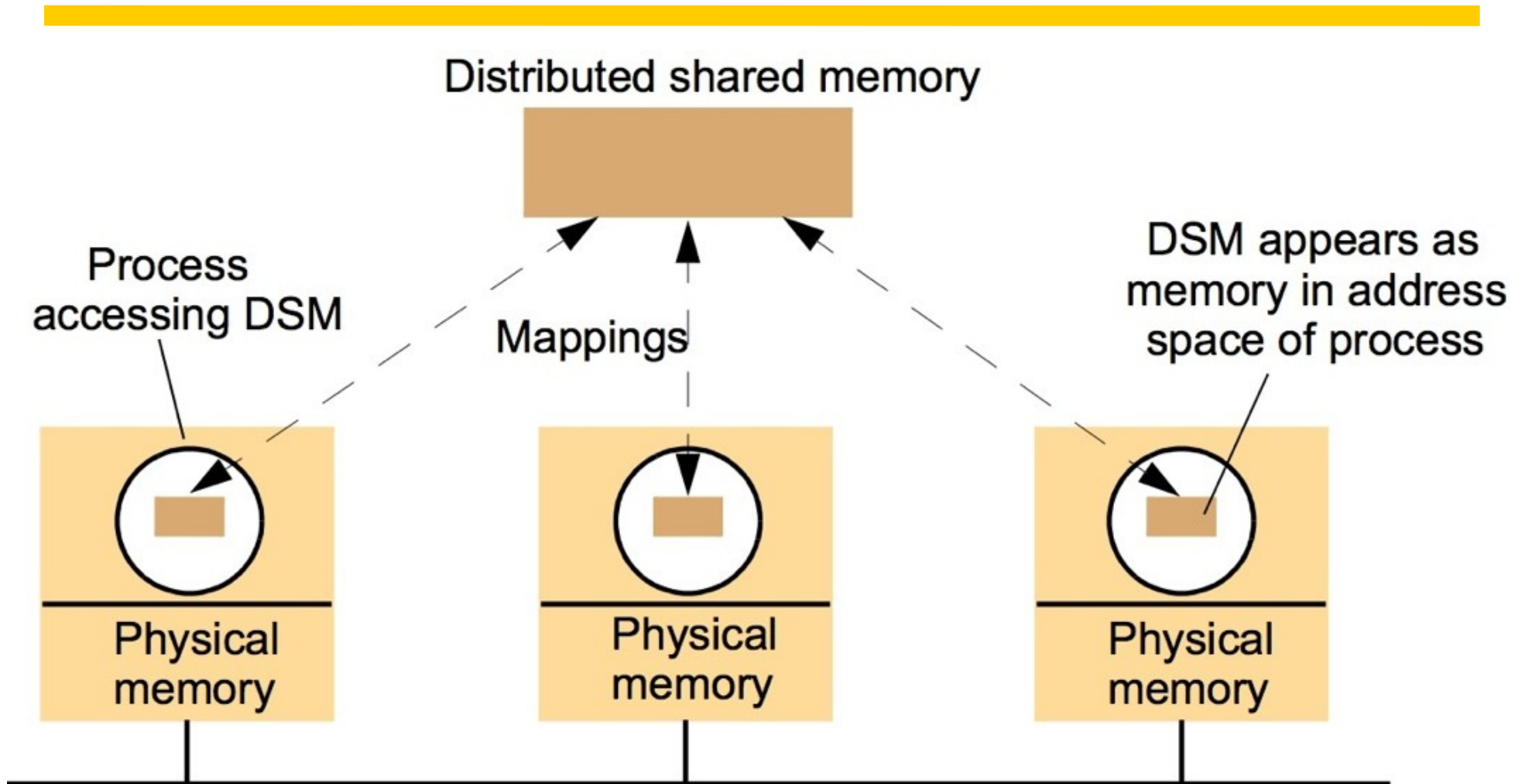


Figure 6.20
The tuple space abstraction

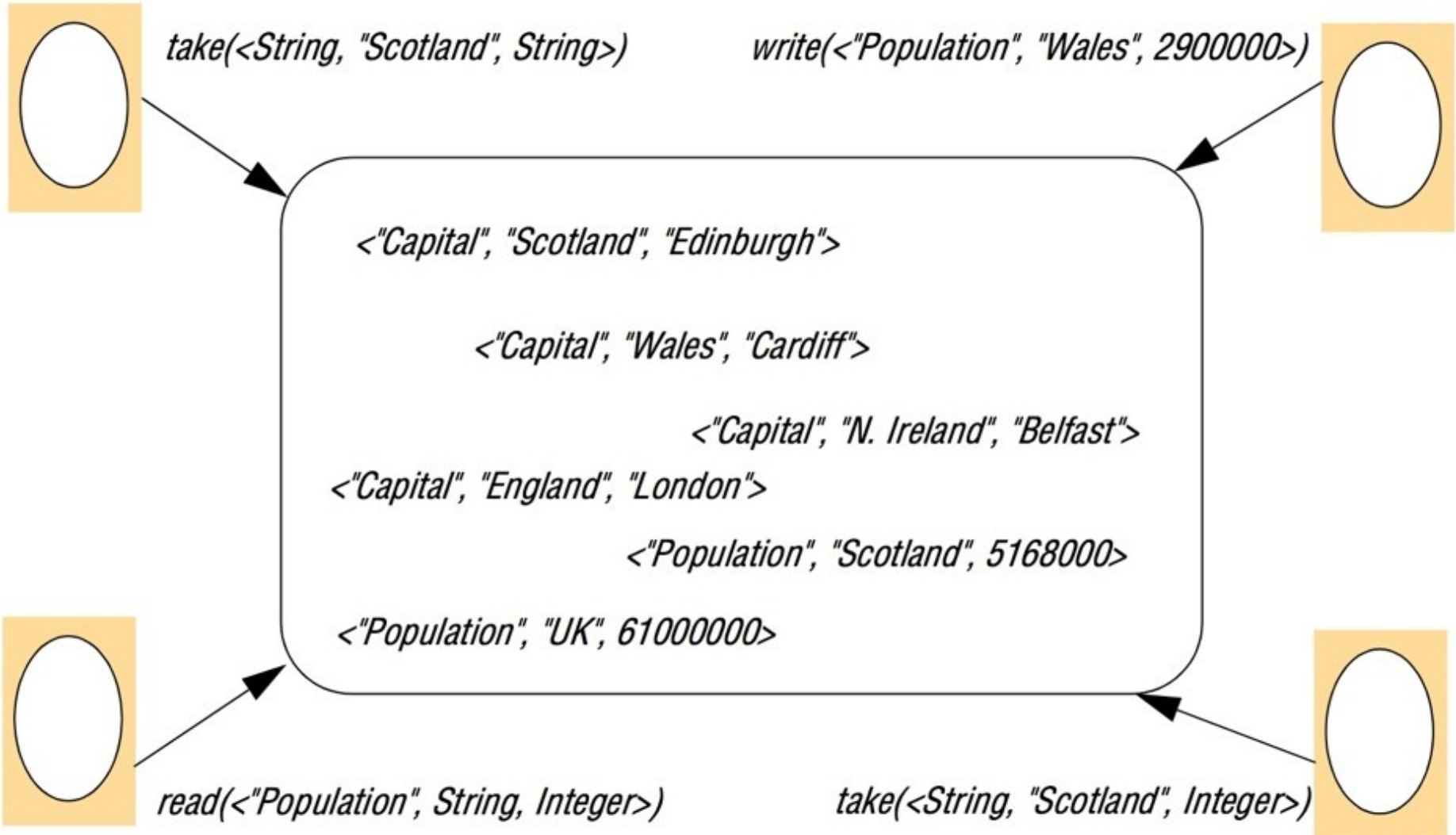


Figure 6.21 (1)

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

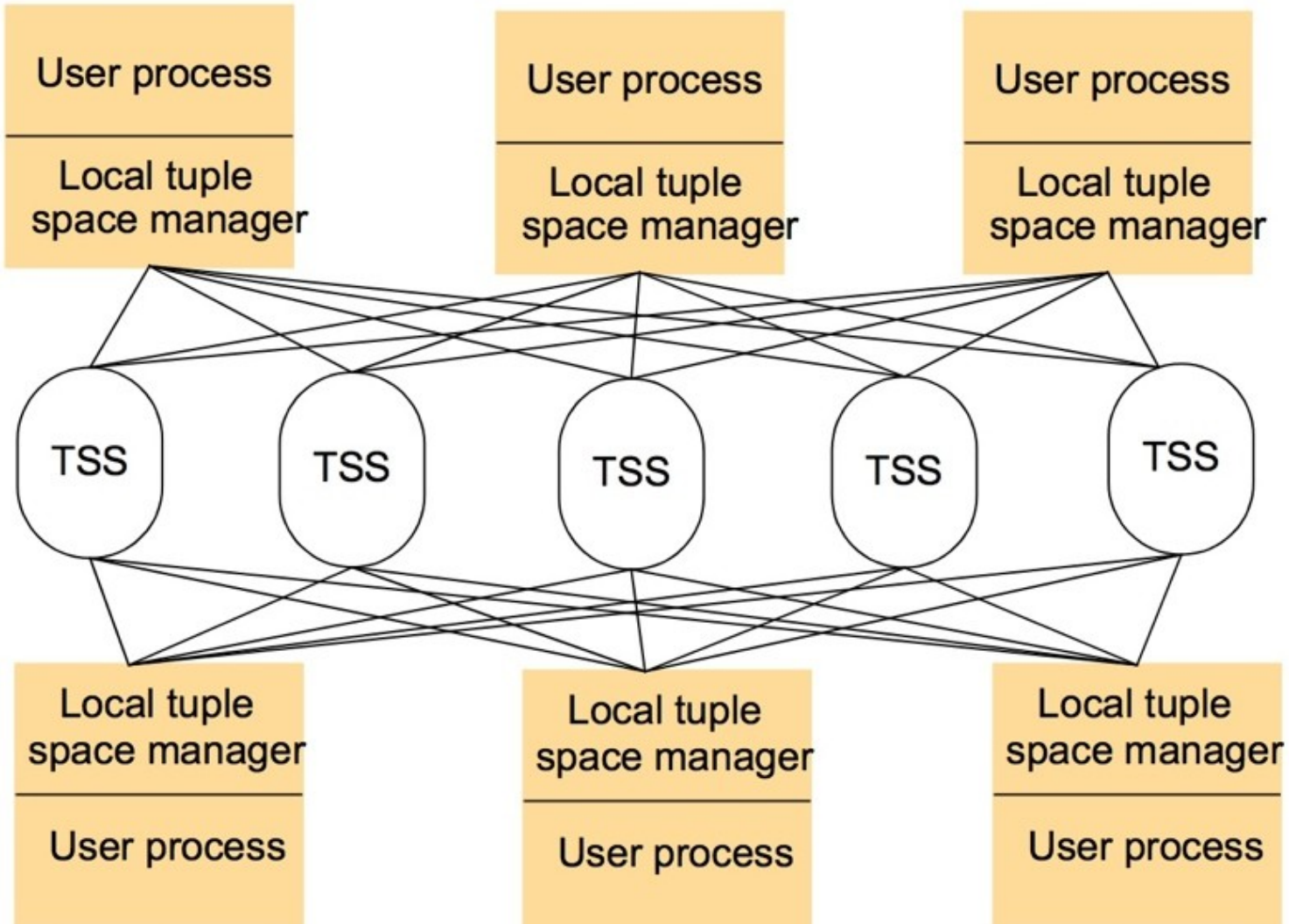


Figure 6.23 The JavaSpaces API

<i>Operation</i>	<i>Effect</i>
<i>Lease write(Entry e, Transaction txn, long lease)</i>	Places an entry into a particular JavaSpace
<i>Entry read(Entry tmpl, Transaction txn, long timeout)</i>	Returns a copy of an entry matching a specified template
<i>Entry readIfExists(Entry tmpl, Transaction txn, long timeout)</i>	As above, but not blocking
<i>Entry take(Entry tmpl, Transaction txn, long timeout)</i>	Retrieves (and removes) an entry matching a specified template
<i>Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)</i>	As above, but not blocking
<i>EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)</i>	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

	<i>Groups</i>	<i>Publish-subscribe systems</i>	<i>Message queues</i>	<i>DSM</i>	<i>Tuple spaces</i>
<i>Space-uncoupled</i>	Yes	Yes	Yes	Yes	Yes
<i>Time-uncoupled</i>	Possible	Possible	Yes	Yes	Yes
<i>Style of service</i>	Communication-based	Communication-based	Communication-based	State-based	State-based
<i>Communication pattern</i>	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
<i>Main intent</i>	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
<i>Scalability</i>	Limited	Possible	Possible	Limited	Limited
<i>Associative</i>	No	Content-based publish-subscribe only	No	No	Yes