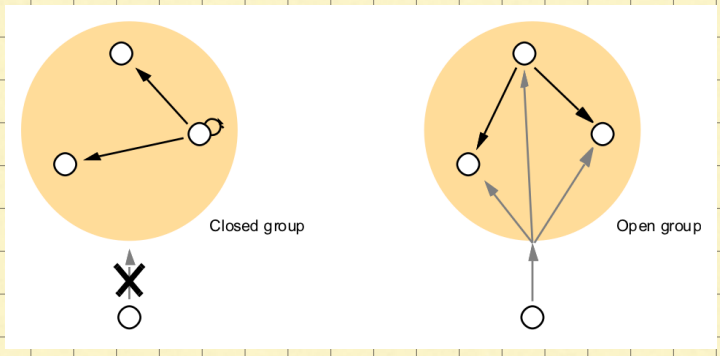
Space and time coupling in distributed systems

	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

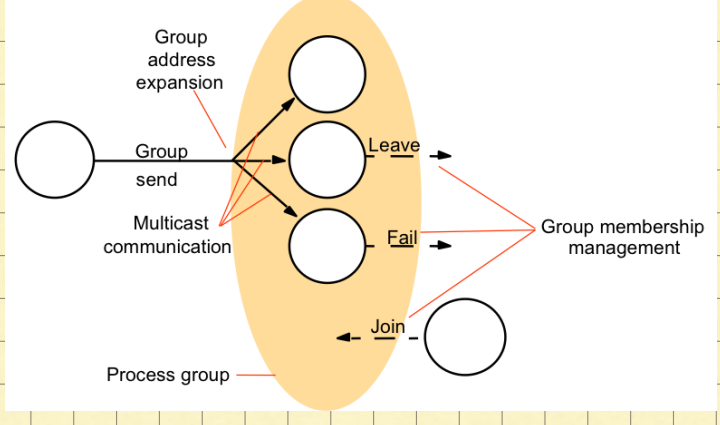
Open and closed groups



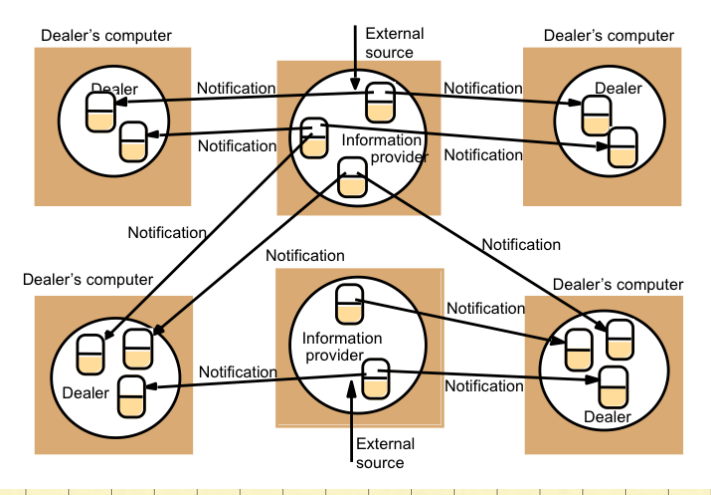
Reliability and Ordering

- Integrity vs validity vs agreement
- Ordering
  - FIFO
  - Causal
  - Total
- Group Management
  - Providing an interface for group membership changes
  - Failure detection
  - Notifying members of group membership changes
  - Performing group address expansion

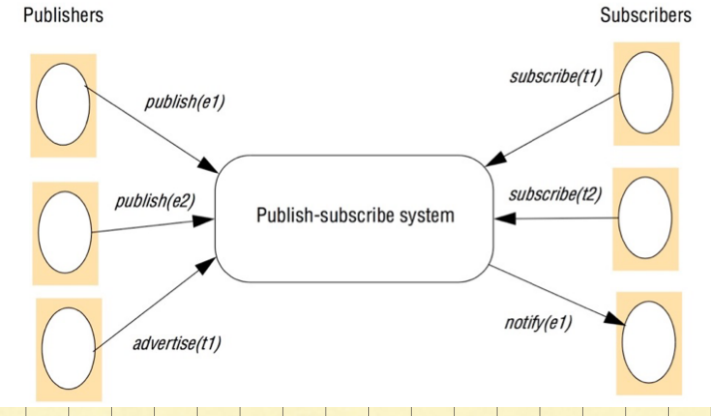
Group membership management



Dealing room system



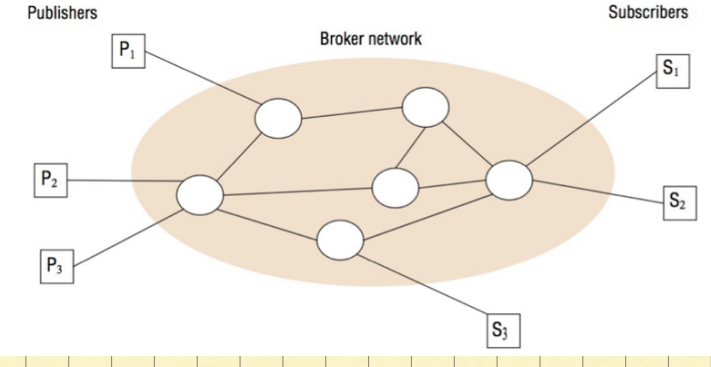
The publish-subscribe paradigm



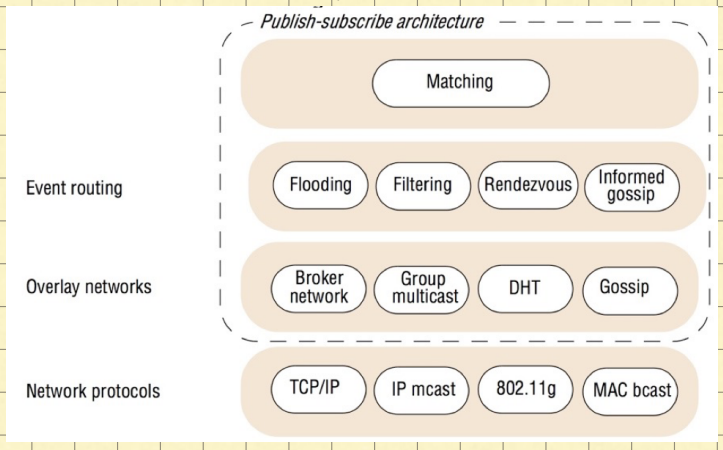
- Heterogeneity
- Asynchronicity
- Channel based
- Topic based
- Content based
- Type based

Implementation Issues

- Centralized versus distributed implementations



Architecture of publish-subscribe systems



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
```

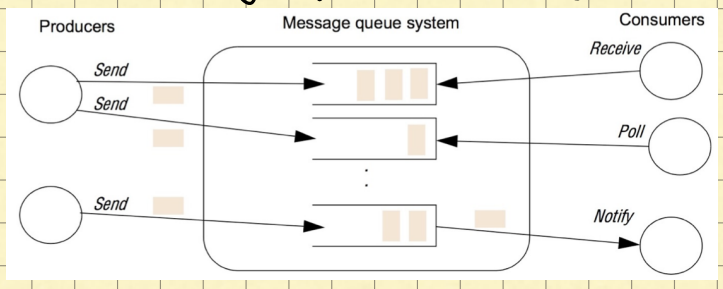
Rendezvous-based routing

```
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;
```

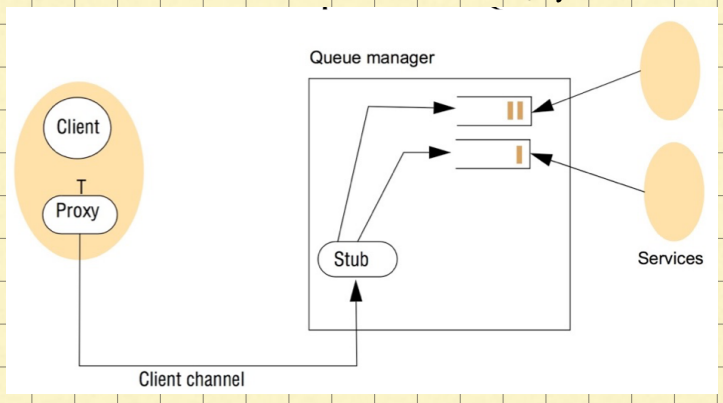
Examples

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	Flooding
Meghdoot [Gupta *et al.* 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni *et al.* 2005]	Content-based	Peer-to-peer	Informed gossip

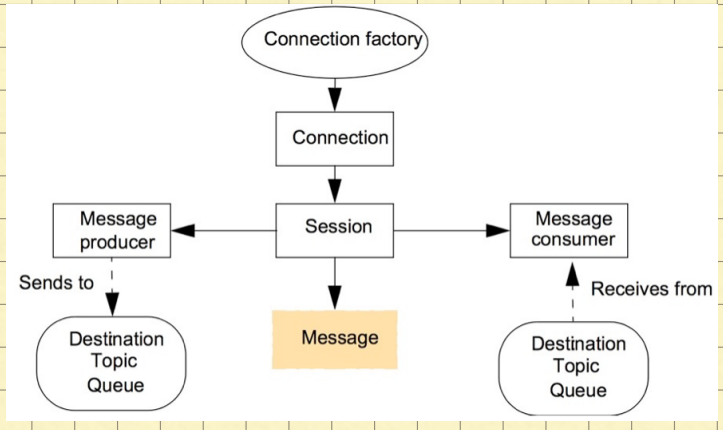
The message queue paradigm



Simple networked topology in WebSphere MQ



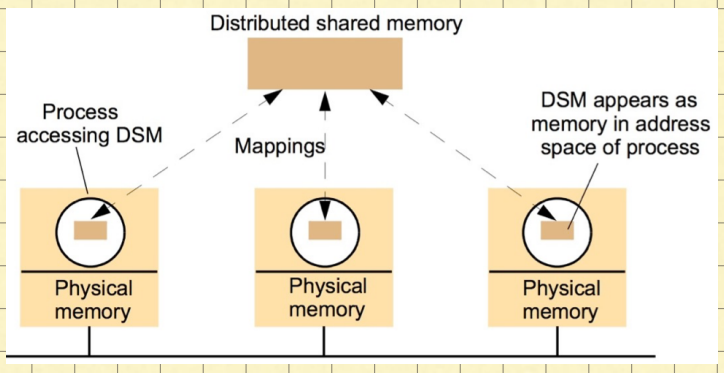
Programming model offered by JMS



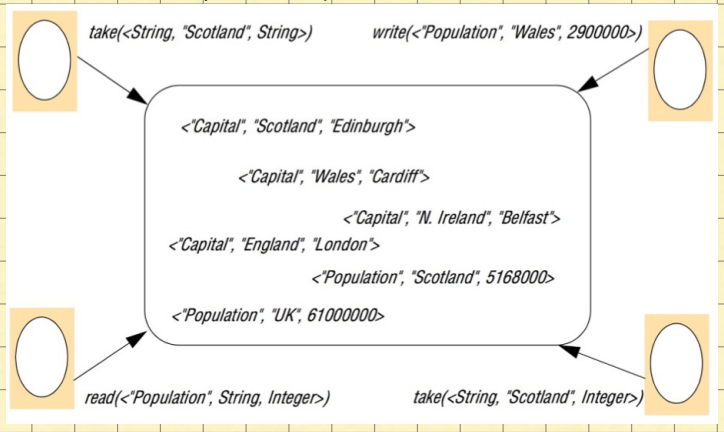

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {

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
}
```

```
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
}
```

The distributed shared memory abstraction



The tuple space abstraction



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

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

```
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) {
    }
   }
}
```

```
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;
        }
    }
}
```

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