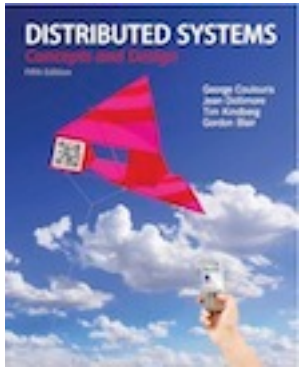
Slides for Chapter 6:
Indirect Communication



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

Distributed Systems:

Concepts and Design

Edition 5, © Addison-Wesley 2012

Space and time uncoupling

Space uncoupling: the sender does not know (or need to know) the identity of the receiver(s).

Time uncoupling: the sender and receiver(s) can have independent lifetimes (i.e., the sender and the receiver does not need to exist at the same time to communicate)

"All problems in computer science can be solved by another level of indirection" (R. Needham)

(but "there is no performance problem that cannot be solved by eliminating a level of indirection" (J. Gray))

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 6.3
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

Group communication

Group communication offers a service whereby a message is sent to a group and then this message is delivered to all members of the group

Characteristics

- Sender is not aware of the identities of the receivers
- Represents an abstraction over multicast communication

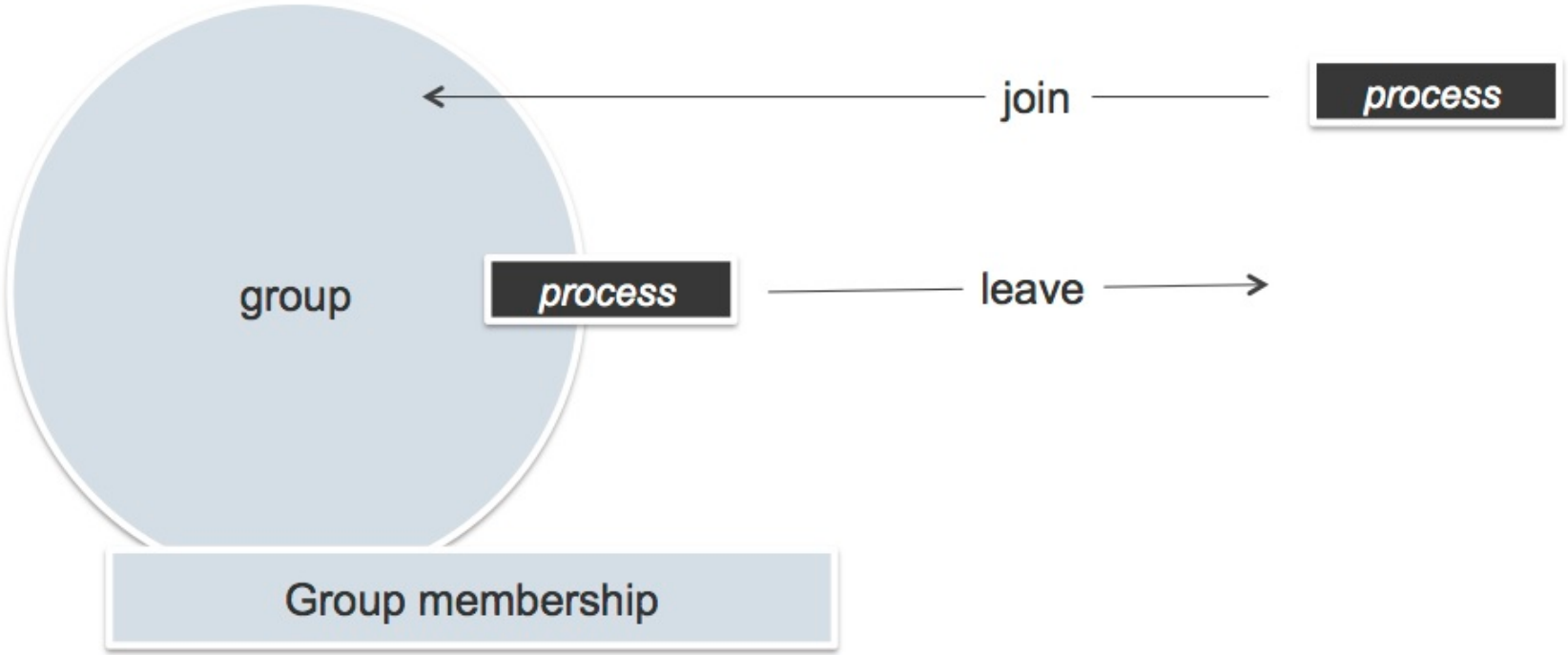
Possible implementation over IP multicast (or an equivalent overlay network), adding value in terms of

- Managing group membership
- Detecting failures and providing reliability and ordering guarantees

Key areas of applications of group communication

- Reliable dissemination of information to potentially large numbers of clients, including financial industry, where institutions require accurate and up-to-date access to a wide variety of information sources
- Support for collaborative applications, where events must be disseminated to multiple users to preserve a common user view – for example, in multiuser games
- Support for a range of fault-tolerance strategies, including the consistent update of replicated data or the implementation of highly available (replicated) servers
- Support for system monitoring and management, including for example load balancing strategies

Programming model



Process groups and object groups

Most group services are suing the concept of **process groups**

- Communicating entities are processes
- Messages are delivered to processes and no further support for dispatching is provided.
- Messages are typically unstructured byte arrays with no support for marshalling or complex data types

Object groups provide a higher-level approach to group computing

- Collection of objects (normally instances of the same class) that process the same set of invocations concurrently, with each returning responses
- Client objects need to be aware of the replication => they invoke operations on a single, local objects which acts as a proxy for the group
- Proxy uses a group communication systems to send the invocations to the members of the object group

Figure 6.2
Open and closed groups

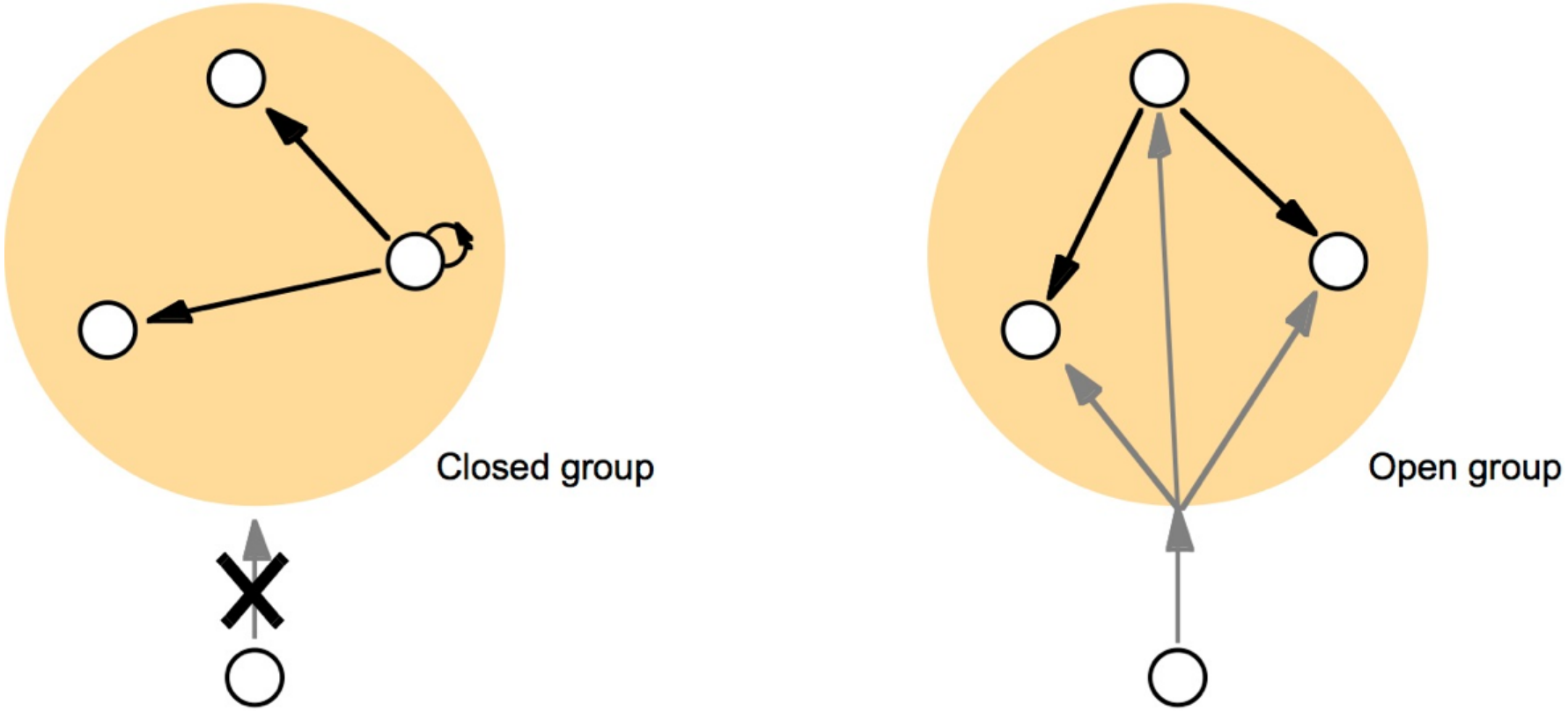
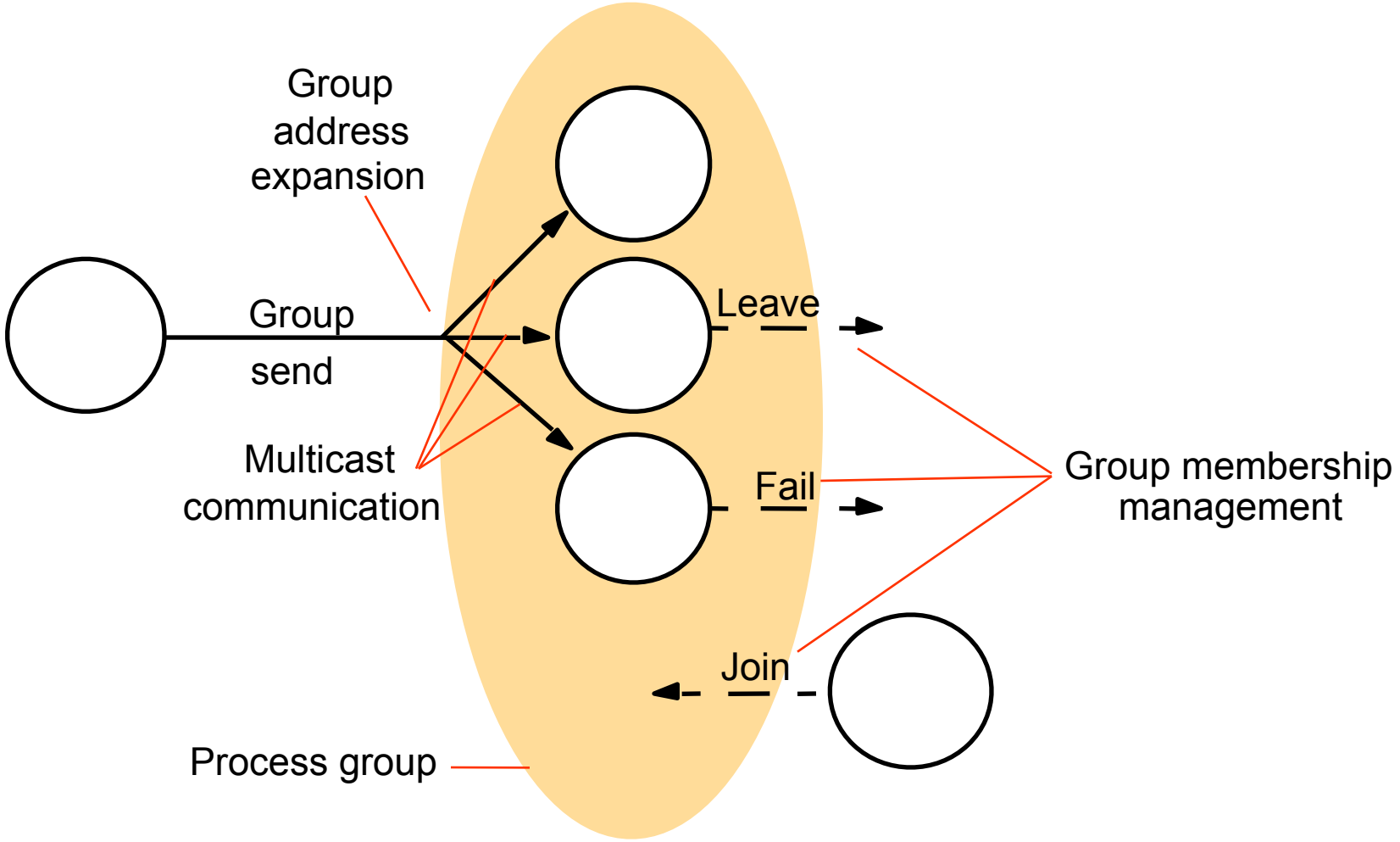


Figure 6.3
The role of group membership management



Group membership management

Overall goal: maintaining an accurate view of the current membership.

Providing an interface for group membership changes

The membership service provides operations to create and destroy process groups

Failure detection

The service monitors the group members not only in case they should crash, but also in case they should become unreachable because of a communication failure.

Notifying members of group membership changes

The service notifies the group's members when a process is added, or when a process is excluded.

Performing group address expansion

When a process multicasts a message, it supplies the group identifier rather than a list of processes in the group.

Group membership management

- Maintaining group membership has significant impact on the utility group-based approaches
  - most effective in small-scale and static systems
  - does not operate as well in larger scale, and highly volatile systems
- Can be seen as a form of *synchrony* assumption
- Probabilistic approaches for group membership in large scale groups, using gossip protocol
- Also specific protocols for ad hoc networks and mobile environments

Reliability in one-to-one communication

Integrity: the message received is the same as the one sent, and no messages are delivered twice

Validity: any outgoing message is eventually delivered

Reliable Multicast

Integrity: delivering the messages to each receiver correctly, and at most once

Validity: guaranteeing that a message sent will eventually be delivered

(+) Agreement: Stating that if the message is delivered to one process, then it is delivered to all processes in the group

Ordering in multicast

First-in-first-out (FIFO) ordering is concerned with preserving the order from the perspective of a sender process, in that if a process sends one message before another, it will be delivered in this order to all processes in the group

Causal ordering takes into account causal relationships between the messages, in that if a message *happens before* another message in the distributed system this *causal* relationship will be preserved in the delivery of the associated message at all processes.

In **total ordering**, if a message is delivered before another message at one process, then the same order will be preserved at all processes.

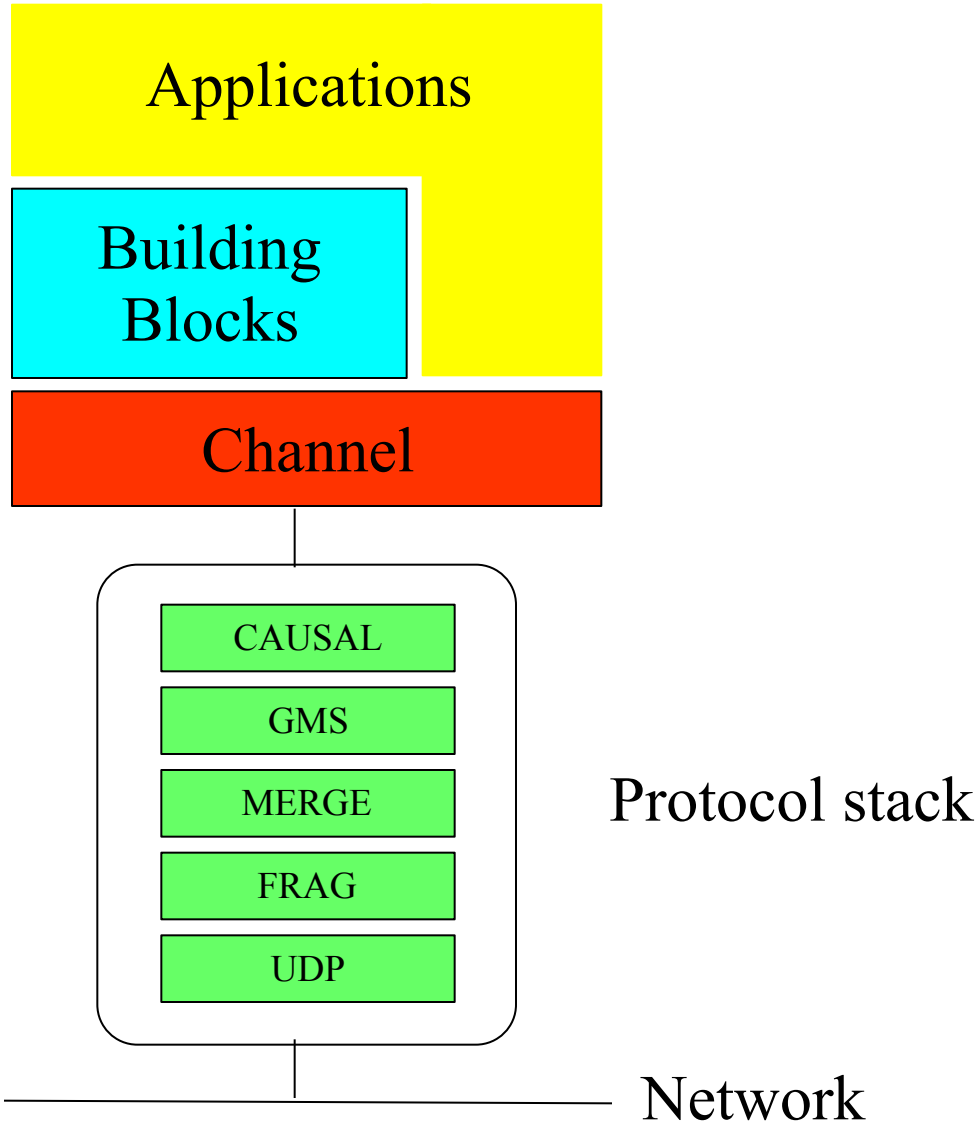
JGroups (www.jgroups.org)

- JGroups is a **Java toolkit** for reliable multicast communication.
- It allows a Java programmer to create groups of processes whose members can send messages to each other in an easy way.
- Some pleasant features are:
  - **group creation** and **deletion** (spreading across LANs or WANs);
  - **joining** and **leaving** of groups;
  - **membership detection** and **notification** about joined/left/crashed members;
  - **detection** and **removal** of crashed members;
  - sending and receiving of **member-to-group messages** (point-to-multipoint);
  - sending and receiving of **member-to-member messages** (point-to-point)
  - Based on UDP multicast (no server needed)
- The main advantage is that developers can create reliable multicast applications, without having to implement the reliable communication layer.

JGroups overview

- The main class of the JGroups API is **JChannel**.
- Such a class allows
  - to connect to a cluster (i.e., group),
  - to send and receive messages,
  - to register listeners that are called when events (such as member joins) happen.
- Members send around **Messages**, containing a byte buffer (the payload), plus the sender's and receiver's address.
- **Addresses** are subclasses of org.jgroups.Address (usually containing an IP address and a port number).
- The list of instances in a cluster is called a view (org.jgroups.View), and every instance contains exactly the same **View**.
- The list of the addresses of all instances can get retrieved by calling **View.getMembers()**.
- Instances can send/receive messages only when they are in a cluster.
- **JChannel.disconnect()** or **JChannel.close()** can be called when an instance wants to leave the cluster.

JGroups architecture



JGroups main components

- **Channels** act as handles onto a group:
  - when created they are disconnected;
  - a connect operation binds a channel to a named group;
  - a process may leave the group performing a disconnect operation;
  - the close operation render the channel unusable.
- **Building blocks** are abstraction for advanced communication paradigms:
  - MessageDispatcher;
  - RpcDispatcher;
  - NotificationBus.
- The **protocol stack** is bidirectional passing events by means of two methods:
  - public Object up(Event evt);
  - public Object down(Event evt);

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;
       }
   }
}
```

Creating and joining channels (clusters):

```
import org.jgroups.JChannel;

public class SimpleChat extends ReceiverAdapter {
    JChannel channel;
    String user_name=System.getProperty("user.name", "n/a");

    private void start() throws Exception {
        channel=new JChannel();
        channel.connect("ChatCluster");
    }

    public static void main(String[] args) throws Exception {
        new SimpleChat().start();
    }
}
```

Creating and joining a channel

The main event loop and sending chat messages:

```
private void start() throws Exception {
        channel=new JChannel();
        channel.connect("ChatCluster");
        eventLoop();
        channel.close();
    }

    private void eventLoop() {
        BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            try {
                System.out.print("> "); System.out.flush();
                String line=in.readLine().toLowerCase();
                if(line.startsWith("quit") || line.startsWith("exit")) {
                    break;
                }
                line="[" + user_name + "] " + line;
                Message msg=new Message(null, null, line);
                channel.send(msg);
            }
            catch(Exception e) {
            }
        }
}
```

receiver: null = all processes

sender's address

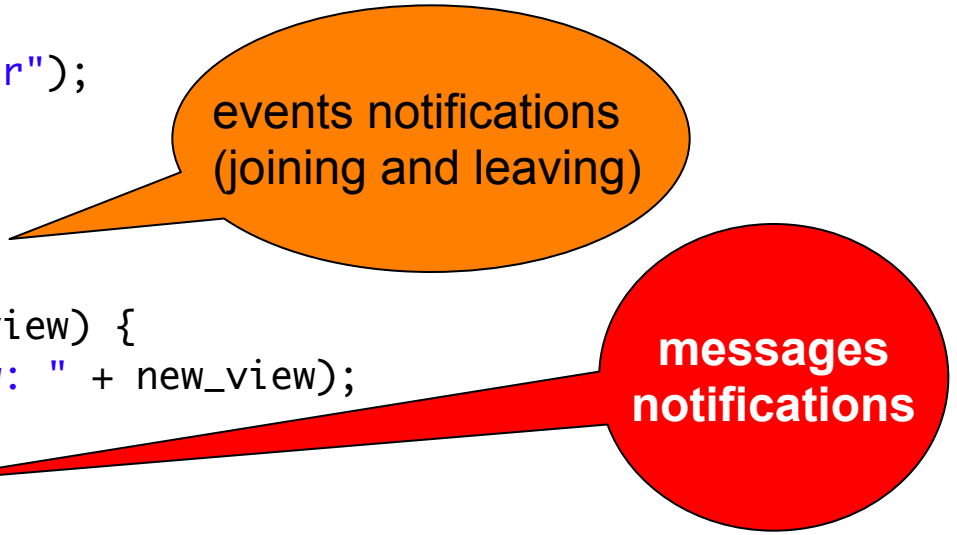
Receiving messages and view change notifications:

```
public class SimpleChat extends ReceiverAdapter {

private void start() throws Exception {
        channel=new JChannel();
        channel.setReceiver(this);
        channel.connect("ChatCluster");
        eventLoop();
        channel.close();
}

public void viewAccepted(View new_view) {
        System.out.println("** view: " + new_view);
}

public void receive(Message msg) {
        System.out.println(msg.getSrc() + ": " + msg.getObject());
}
```



Compilation & execution

- **`export CLASSPATH=$CLASSPATH:`**/path-to/**`JGroups-2.11.0.GA.bin/jgroups-2.11.0.GA.jar`**
- **`javac SimpleChat.java`**
- for each client: **`java SimpleChat`**
- if no network connections are available (loopback only):
- **`java -Djgroups.bind_addr=127.0.0.1`**
  `-Djava.net.preferIPv4Stack=true SimpleChat`

Sample run (1)

Start process 1:

```
-------------------------------------------------------------------
GMS: address=rfc-1918-34912, cluster=ChatCluster, physical address=fe80:0:0:0:21c:42ff:fe00:9%8:57585
-------------------------------------------------------------------
** view: [rfc-1918-34912|0] [rfc-1918-34912]
>
```

Start process 2:

```
-------------------------------------------------------------------
GMS: address=rfc-1918-42756, cluster=ChatCluster, physical address=fe80:0:0:0:21c:42ff:fe00:9%8:61433
-------------------------------------------------------------------
** view: [rfc-1918-34912|1] [rfc-1918-34912, rfc-1918-42756]
>
```

Process 1 is notified:

```
-------------------------------------------------------------------
GMS: address=rfc-1918-34912, cluster=ChatCluster, physical address=fe80:0:0:0:21c:42ff:fe00:9%8:57585
-------------------------------------------------------------------
** view: [rfc-1918-34912|0] [rfc-1918-34912]
> ** view: [rfc-1918-34912|1] [rfc-1918-34912, rfc-1918-42756]
```

Sample run (2)

Sending a message:

```
--------------------------------------------------------------------
GMS: address=rfc-1918-34912, cluster=ChatCluster, physical address=fe80:0:0:0:21c:42ff:fe00:9%8:57585
--------------------------------------------------------------------
** view: [rfc-1918-34912|0] [rfc-1918-34912]
> ** view: [rfc-1918-34912|1] [rfc-1918-34912, rfc-1918-42756]
hello, world!
> rfc-1918-34912: [ivan] hello, world!
```

Message is broadcasted:

```
--------------------------------------------------------------------
GMS: address=rfc-1918-42756, cluster=ChatCluster, physical address=fe80:0:0:0:21c:42ff:fe00:9%8:61433
--------------------------------------------------------------------
** view: [rfc-1918-34912|1] [rfc-1918-34912, rfc-1918-42756]
> rfc-1918-34912: [ivan] hello, world!
```

JGroups

- This library enforces the following properties:
  - messages are sent/received in views;
  - each member has a view (i.e. a set of processes that constitute the current membership);
  - when the membership changes, a new view will be installed by all members;
  - **views** are installed in the **same order** at all members;
  - the set of **messages** between **2 consecutive views** will be the **same** at all receivers;
  - messages sent in a given view will be received by all non-faulty members in such view.
- Projects on distributed systems topics based on JGroups:
  - Memcached on JGroups
  - A clustered task distribution system
  - ReplCache: variable caching in the cloud
  - Other projects: http://www.jgroups.org/open_projects.html

JGroups (sample configuration file for UDP protocol)

```
<config>
<UDP mcast_addr="${jgroups.udp.mcast_addr:228.10.10.10}"
     mcast_port="${jgroups.udp.mcast_port:45588}" discard_incompatible_packets="true"
     max_bundle_size="60000"
     max_bundle_timeout="30"
     ip_ttl="${jgroups.udp.ip_ttl:2}" enable_bundling="true" thread_pool.enabled="true"
     thread_pool.min_threads="1" thread_pool.max_threads="25"
     thread_pool.keep_alive_time="5000" thread_pool.queue_enabled="false"
     thread_pool.queue_max_size="100" thread_pool.rejection_policy="Run"
     oob_thread_pool.enabled="true" oob_thread_pool.min_threads="1"
     oob_thread_pool.max_threads="8" oob_thread_pool.keep_alive_time="5000"
     oob_thread_pool.queue_enabled="false" oob_thread_pool.queue_max_size="100"
     oob_thread_pool.rejection_policy="Run"/>
<PING timeout="2000" num_initial_members="3"/>
<MERGE2 max_interval="30000" min_interval="10000"/>
<FD_SOCK/>
<FD timeout="10000" max_tries="5" />
<VERIFY_SUSPECT timeout="1500" /> <BARRIER />
<pbcast.NAKACK use_mcast_xmit="false" gc_lag="0" retransmit_timeout="300,600,1200,2400,4800"
discard_delivered_msgs="true"/>
<UNICAST timeout="300,600,1200,2400,3600"/>
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000" max_bytes="400000"/>
<VIEW_SYNC avg_send_interval="60000" />
<pbcast.GMS print_local_addr="true" join_timeout="3000" view_bundling="true"/>
<FC max_credits="20000000" min_threshold="0.10"/> <FRAG2 frag_size="60000" />
<pbcast.STATE_TRANSFER />
</config>
```

Reliable Communication

Group membership

Failure detection

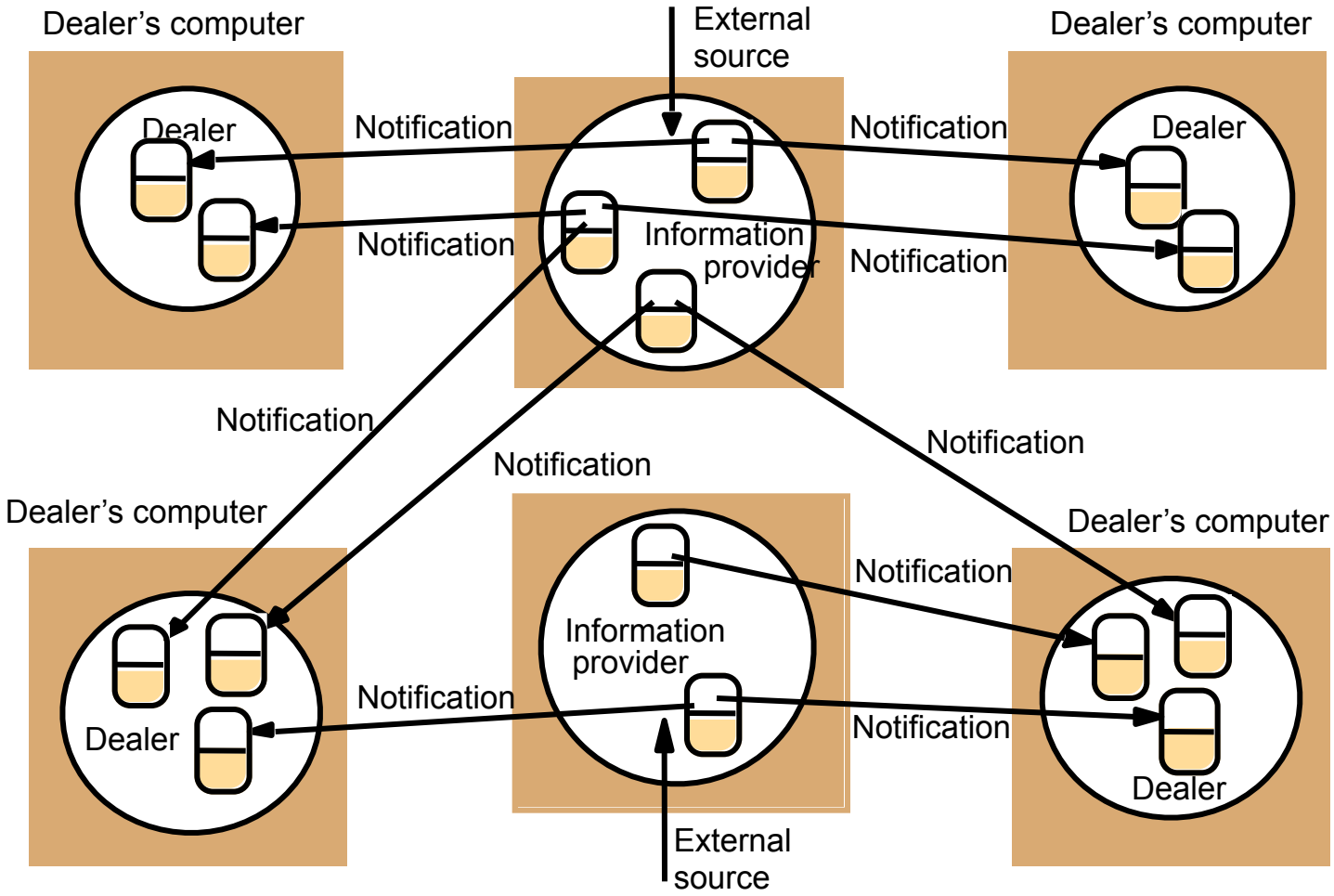
Publish/Subscribe

- **Pub-sub** AKA distributed event systems
  - Most widely used from this chapter
  - Publishers **publish structured events** to event service (ES)
  - Subscribers **express interest** in particular events
  - ES matches published events to subscriptions
- Applications (lots...)
  - Financial info systems
  - Other live feeds of real-time data (including RSS)
  - Cooperative working (events of shared interest)
  - Ubiquitous computing (location events, ... from infrastructure)
  - Lots of monitoring applications, including internet net. mon.
  - Key part of Google infrastructure (‘ad clicks’)

Example: dealing room system

- Example: dealing room for stock trading
  - Let users see latest market prices of stock they care about
  - Info for a given stock arrives from multiple sources
  - Dealers only care about stocks they own (or might)
  - May only care to know above some threshold, in addition
- Possible structure: two (kinds of) tasks
  - Info provider process receives updates (events) from a single external source
  - Dealer process creates subscription for each stock its user(s) express interest in

Figure 6.7
Dealing room system



Characteristics of pub-sub systems

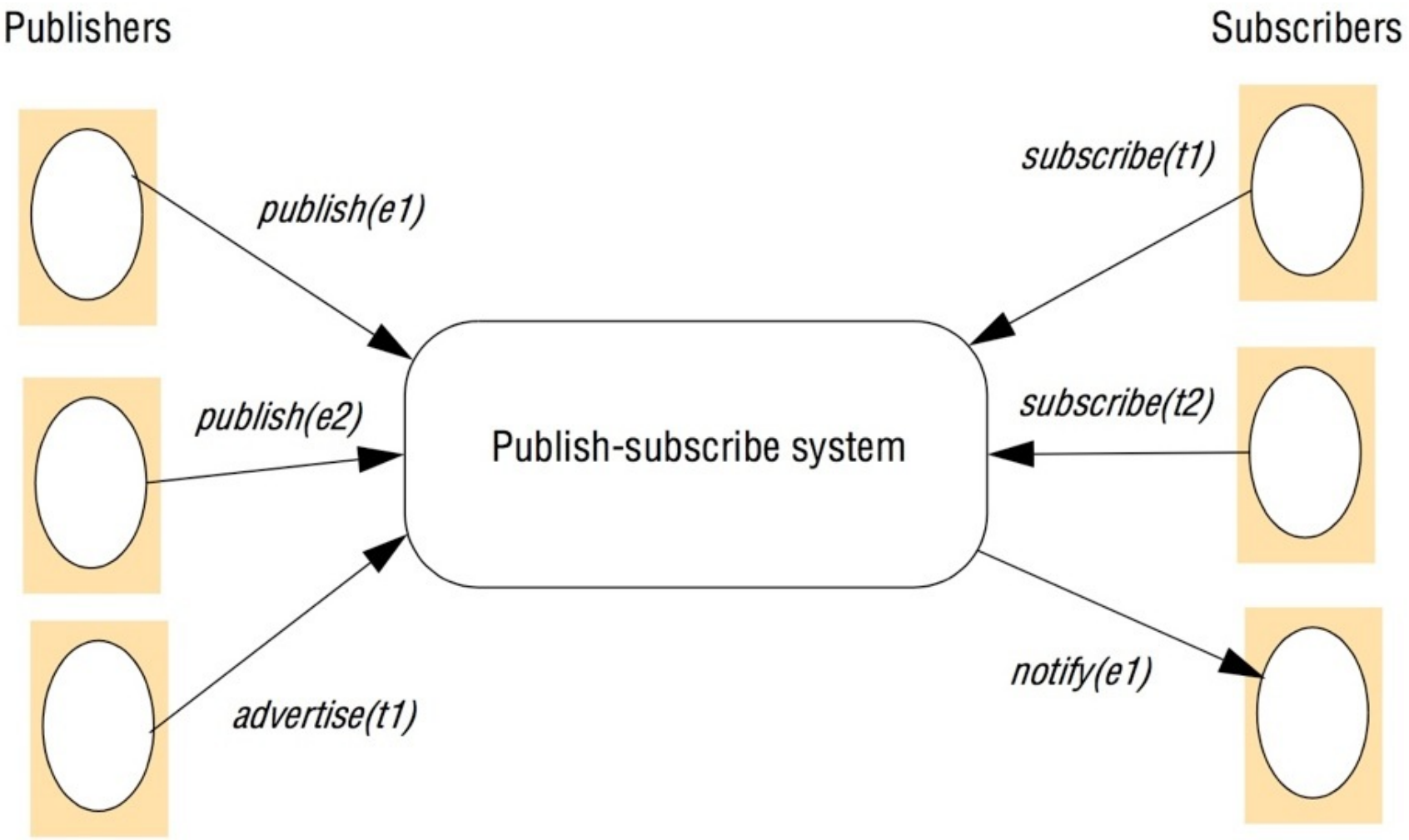
- Heterogeneity
  - Able to glue together systems not designed to work together, with pub-sub technology
  - Have to come up with an external description of what can be subscribed to: simple flat, rich taxonomy, etc
- Asynchrony
  - Decoupling means you never have to block!
- Delivery guarantees (various)
  - All subscribers receive all events (atomicity)
  - Real-time
  - ...

Pub-sub programming model

- Publishers
  - Disseminate event e through `publish(e)`
  - (Sometimes, fancier) register/advertise via a *filter* (pattern over all events) f : `advertise(f)`
    - defines set of notifications the publisher can produce
    - useful to ease routing and reduce the information flow
  - Expressiveness of pattern is the subscription model (later slide)
  - Can also remove the offer to publish: `unadvertise(f)`
- Subscribers
  - Subscribe via a filter (pattern) f: `subscribe(f)`
  - Receive event e matching f: `notify(e)`
  - Cancel their subscription: `unsubscribe(f)`

Figure 6.8

The publish-subscribe paradigm



Subscription models of pub-sub systems

- Channel-based
  - Publishers publish to named channels
  - Subscribers get ALL events from channel
  - Very simplistic, no filtering (all other models below do)
  - CORBA Event Services uses this
- Topic-based (AKA subject-based)
  - Each notification expressed in multiple fields, one being topic
  - Subscriptions choose topics
  - Hierarchical topics can help (e.g., old USENET rec.sports.cricket)

Subscription models of pub-sub systems (cont.)

- Content-based
  - Generalization of topic based
  - Subscription is expression over range of fields (constraints on values)
  - Far more expressive than channel-based or topic-based
- Type-based
  - Use object-based approaches with object types
  - Subscriptions defined in terms of types of events
  - Matching in terms of types or subtypes of filter
  - Coarse grained (type names) to fine grained (attributes and methods of object)
  - Advantage: clean integration with object-based programming languages

Subscription models of pub-sub systems (cont.)

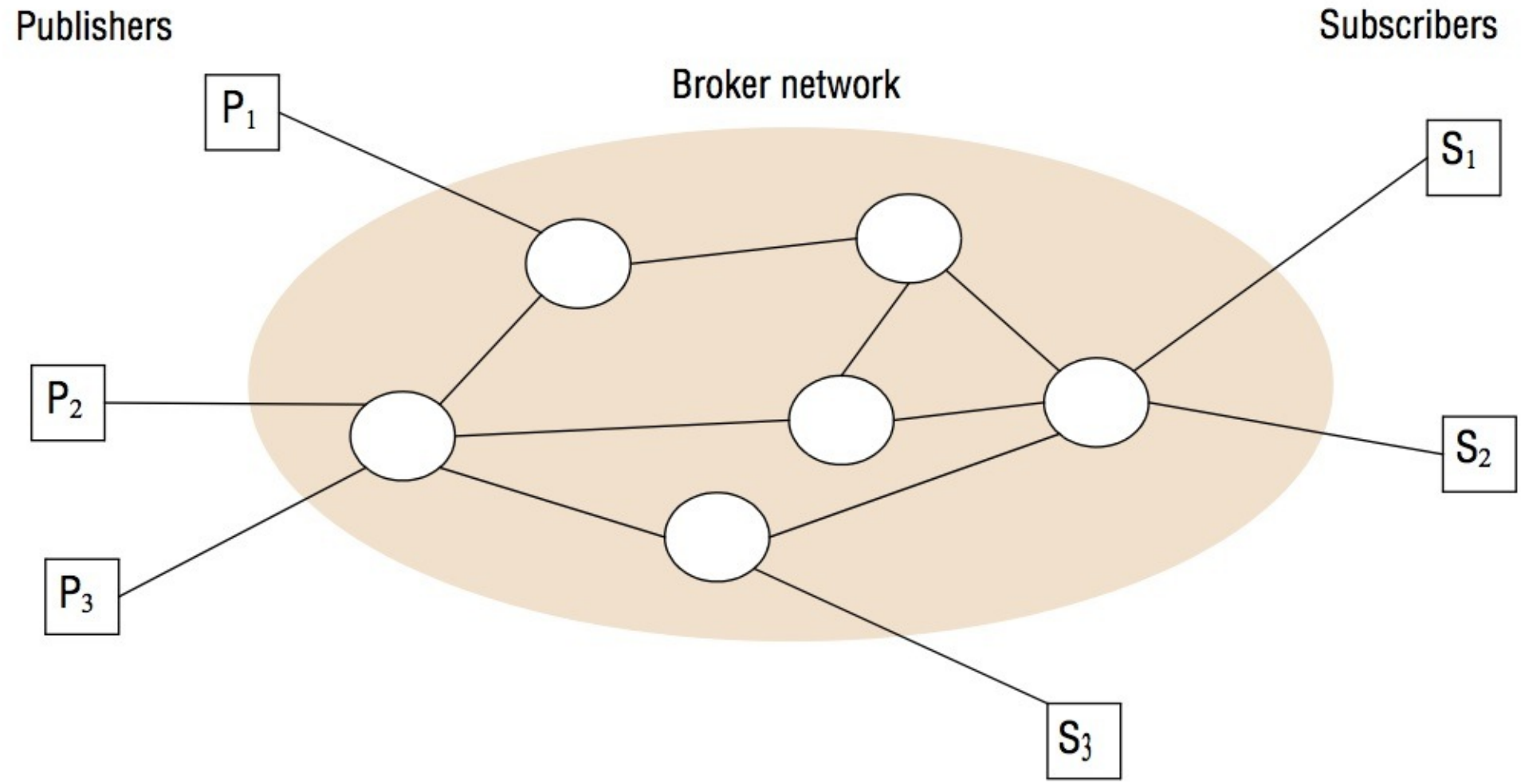
- Objects of interest: like type-based, but on change in state of object
- For mobile: also match based on context
- Concept-based subscriptions: not just syntax, but semantics of events
- Fancier (e.g., financial trading): **complex event processing** (CEP)
  - Patterns between different events, locations, time, ..
  - I.e. patterns can be logical, temporal, or spatial

Implementation issues

- Many ways to delivery events efficiently to subscribers
- Also can be requirements for security, scalability, failure handling, concurrency, QoS
- A number of key implementation choices follow…
- **Centralized vs. distributed implementations**
  - Simple way: single centralized broker node (Limitations?)
  - Most implementations are network of brokers (e.g., GridStat, SIENA)
  - Some implementations are peer-to-peer (P2P)
    - All publisher and subscriber nodes act as the pub-sub broker (e.g., RTI DDS)
  - Q: Plusses and minuses of network of brokers vs. P2P?

Figure 6.9

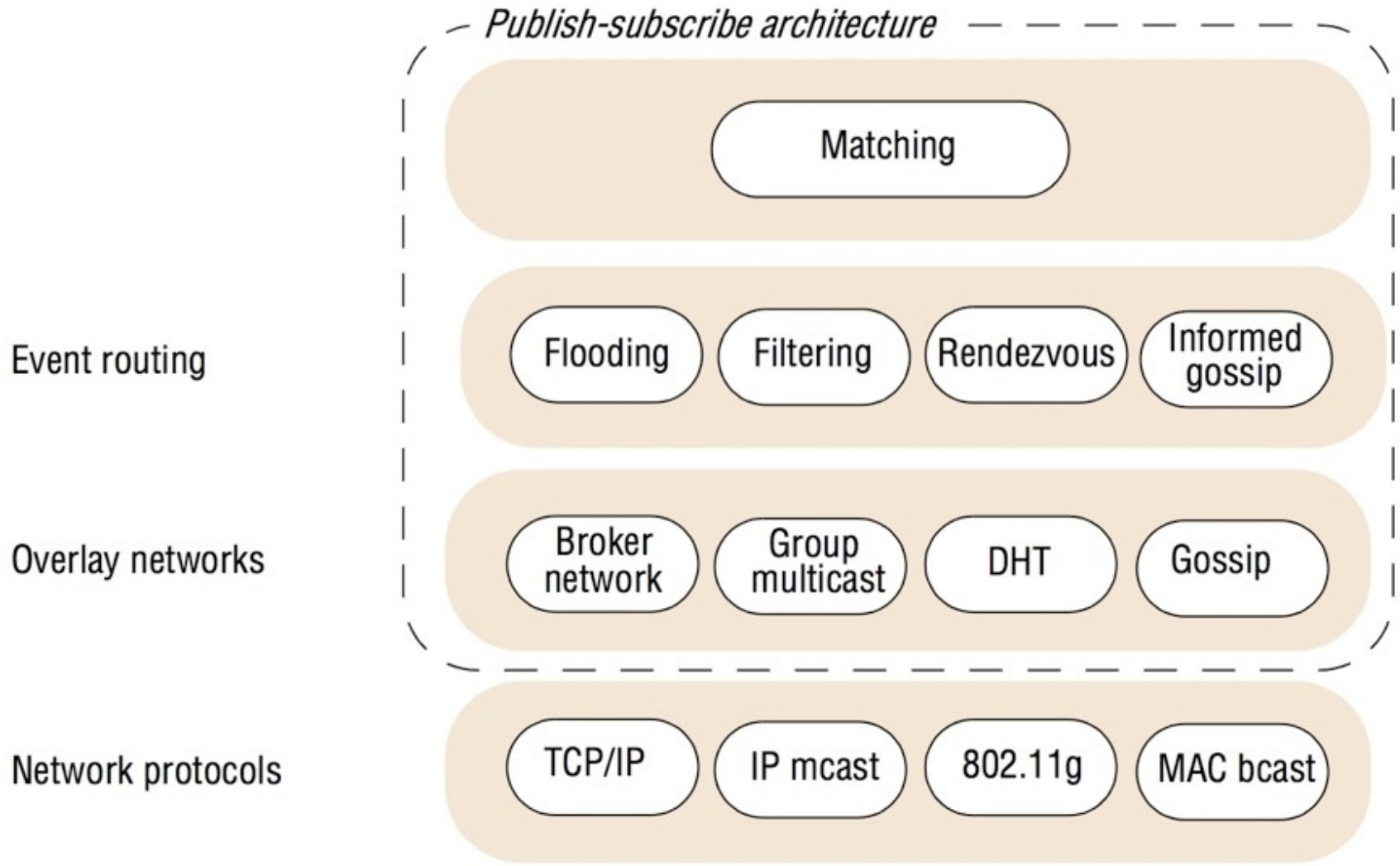
A network of brokers



Overall systems architecture

- Centralized schemes simple...
- Implementing channel-based or topic-based simple
- Map channels/topics onto groups
- Use the group's multicast (possibly reliable, ordered, …)
- Implementation of content/type/ more complicated
- Ranges of choices follow in fig 6.10

Figure 6.10
The architecture of publish-subscribe systems



Implementation choices in content-based routing (CBR)

- Flooding (with duplicate suppression)
  - Simplest version
    - Send event to all nodes on a network
    - Can use underlying multicast/broadcast
  - More complicated
    - Brokers arranged in acyclic forwarding graph
    - Each node forwards to all its neighbors (except one that sent it to node)
- Filtering (filter-based routing)
  - Only forward where path to valid subscriber
  - I.e., subscription info propagated through network towards publ's
    - Each node maintain neighbors list
    - For each neighbor, maintain subscription list/criteria
    - Routing table with list of neighbors and subscribers downstream

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
```

Implementation choices in content-based routing (CBR)

- Advertisements
  - propagate advertisements towards subs' (symmetrical to filtering)
- Rendezvous (Fig 6.12)
  - Consider set of possible events as an event space
  - Partition event space among brokers in net. (rendezvous nodes)
  - SN(s): for given subscrip. s, returns set of nodes responsible for it
  - EN(e): for event e, rtn list of nodes that match e against subscriptions
  - Mapping intersection rule: SN(s)∩ EN(e) must be nonempty if e matches s
  - Distributed hash table (DHT) variant: map events and subscriptions onto a rendezvous nodes via DHT (Sec 4.5.1)
- Routing can be done via *gossiping* (epidemic multicast)

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	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

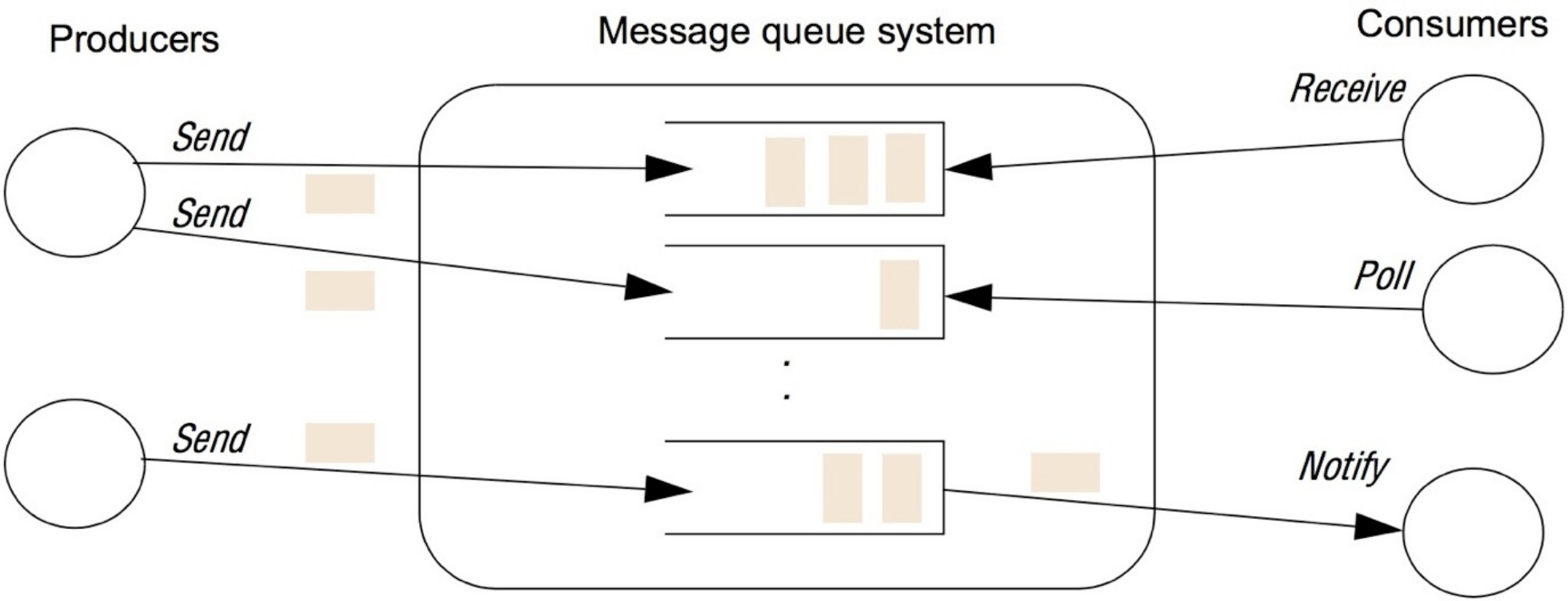
More recent: Redis, Kafka, Google Pub/Sub, Attribute-based Updates

Message queues

- (Distributed) message queues: intermediary between producers and consumers of data
  - Point-to-Point, not one-to-many
  - Supports time and space uncoupling
  - AKA Message-Oriented Middleware (MOM)
  - LOTS of commercial products
  - Common use: **Enterprise Application Integration** (EAI)
  - Also a lot for transactions (6.4.1)
- Programming model: producer sends msg; consumers can
  - Blocking receive
  - Non-blocking receive (polling)
  - Notify

Figure 6.14

The message queue paradigm



Programming model

- Many processes can send to a queue, many can remove from it
- Queuing policy: usually FIFO, but also priority-based
- Consumers can select based on metadata
- Database integration common use; e.g. Oracle Advance Queuing
  - Messages are a row in a (relational) database
  - Queues are database tables that can be SQL-queried against
- Messages are persistent
  - Store until removed
  - Store on a disk
- Other common functionality
  - Transaction support: all-or-none operations
  - Automatic message transformation: on arrival, message transforms data from one format to another (data heterogeneity)
  - Security (at least confidentiality)

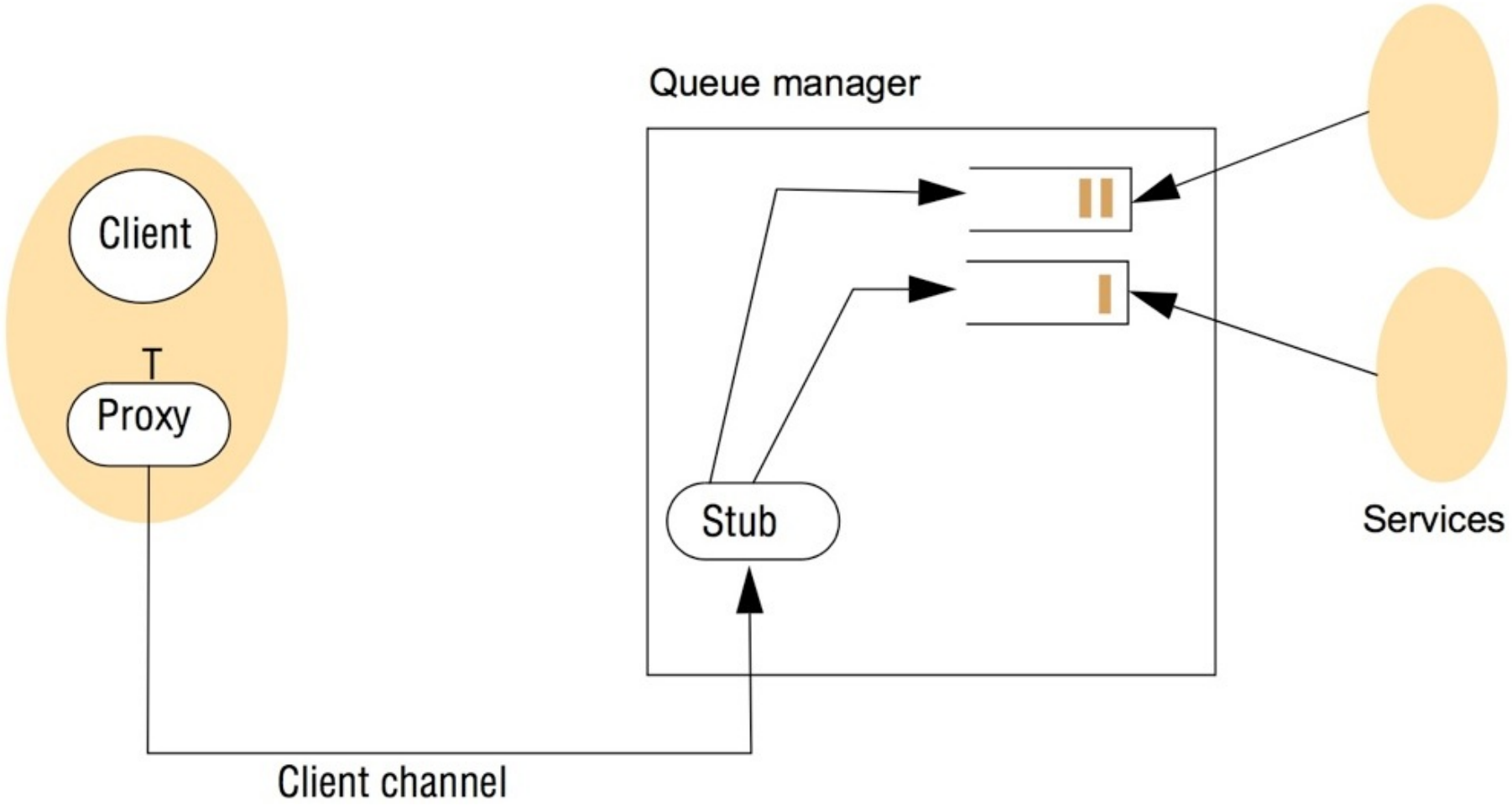
Implementation issues

- Key choice: centralized vs. distributed implementation (tradeoffs?)

- Case study: IBM WebSphere software
  - Queue managers host and manage queues, enable apps to access via Message Queue Interface (MQI)
  - Connect or disconnect to/from a queue
  - Send/receive messages to/from a queue (via a RPC call)
  - Clients not on same host (usual case) via client channel (w/proxy+stub)

Figure 6.15

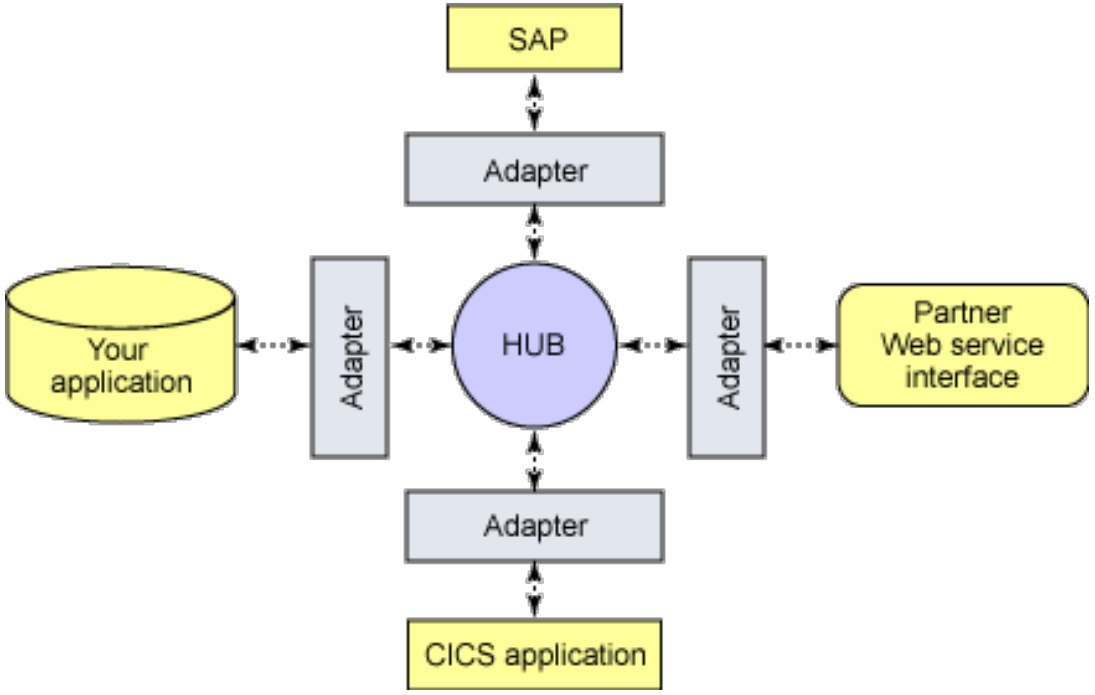
A simple networked topology in WebSphere MQ



IBM WebSphere (cont.)

- Queues usually linked into a *federated structure*
  - Resembles pub-sub, but choose right topology for app
  - Queues linked with message channel (MC)
  - Message channel agent (MCA) manages each end of MC
  - Queue managers have routing tables
  - Lots of tools to create different topologies, manage components, etc
  - E.g.: *Enterprise Integration Patterns*, which can build over Websphere, TIBCO, BizTalk, JMS-based, etc.

- Hub-and-spoke topology (common)
  - Hub has lots of services (and resources to support)
  - Spoke queues are distant, place close(r) to clients
  - Clients interface with spoke queues



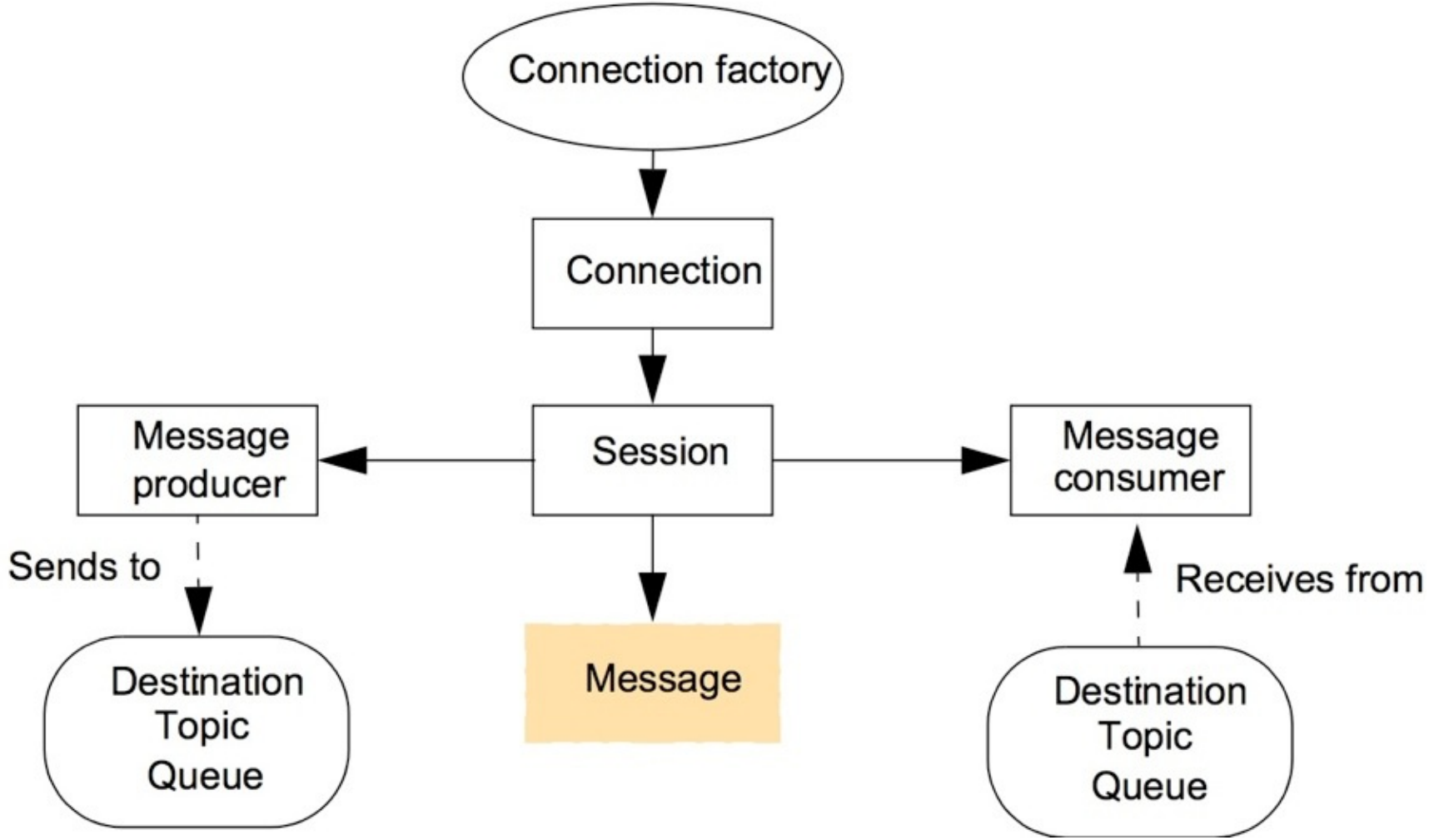
Case study: Java Messaging Service (JMS)

- JMS supports both pub-sub and MQs
  - Many vendors; others provide interface (e.g., WebSphere)
- Key roles in JMS
  - JMS client: Java app that produces or consumes messages
  - JMS producer: creates a message and places in a queue
  - JMS consumer: removes a message from a queue and uses it
  - JMS provider: any system that implements the JMS spec
  - JMS message: object used to communicate between JMS clients
  - JMS destination: object supporting indirect communication in JMS
    - JMS topic: supports pub-sub
    - JMS queue: (um, obvious)

- First create a connection from client to provider with connection factory
  - `TopicConnection` or `QueueConnection`
- Use connection to create ≥1 session
  - Series of operations for creating, producing, consuming msgs for a given logical task
  - Also supports transactions
  - One session can handle topics OR queues, not both

Figure 6.16

The programming model offered by JMS



JMS session objects

- Message has 3 parts
  - Header: everything needed to identify & route msg
    - Destination, priority, expiration date, message ID, timestamp
  - Properties: user-defined meta-data
  - Body: opaque data
- **Message producer**: object that publishes messages to a topic or sends to a queue
- **Message consumer**: subscribe to topics or receive from Q
  - Can associate filters w/consumer: specify a *message selector* (subset of SQL)
- Two modes for receiving messages
  - Block with receive operation
  - Create message listener object with a *callback object* onMessage

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(msg);  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
}
```

AMQP - Advanced Message Queueing Protocol

- Not a set of API, but a protocol
- http://www.amqp.org
- Broadly applicable for enterprise
- Totally open
- Platform agnostic
- Interoperable (language independent)
- Standard port is 5672/tcp
- List of brokers: http://jira.amqp.org/confluence/display/AMQP/AMQP+Products
- In particular: RabbitMQ (written in Erlang)

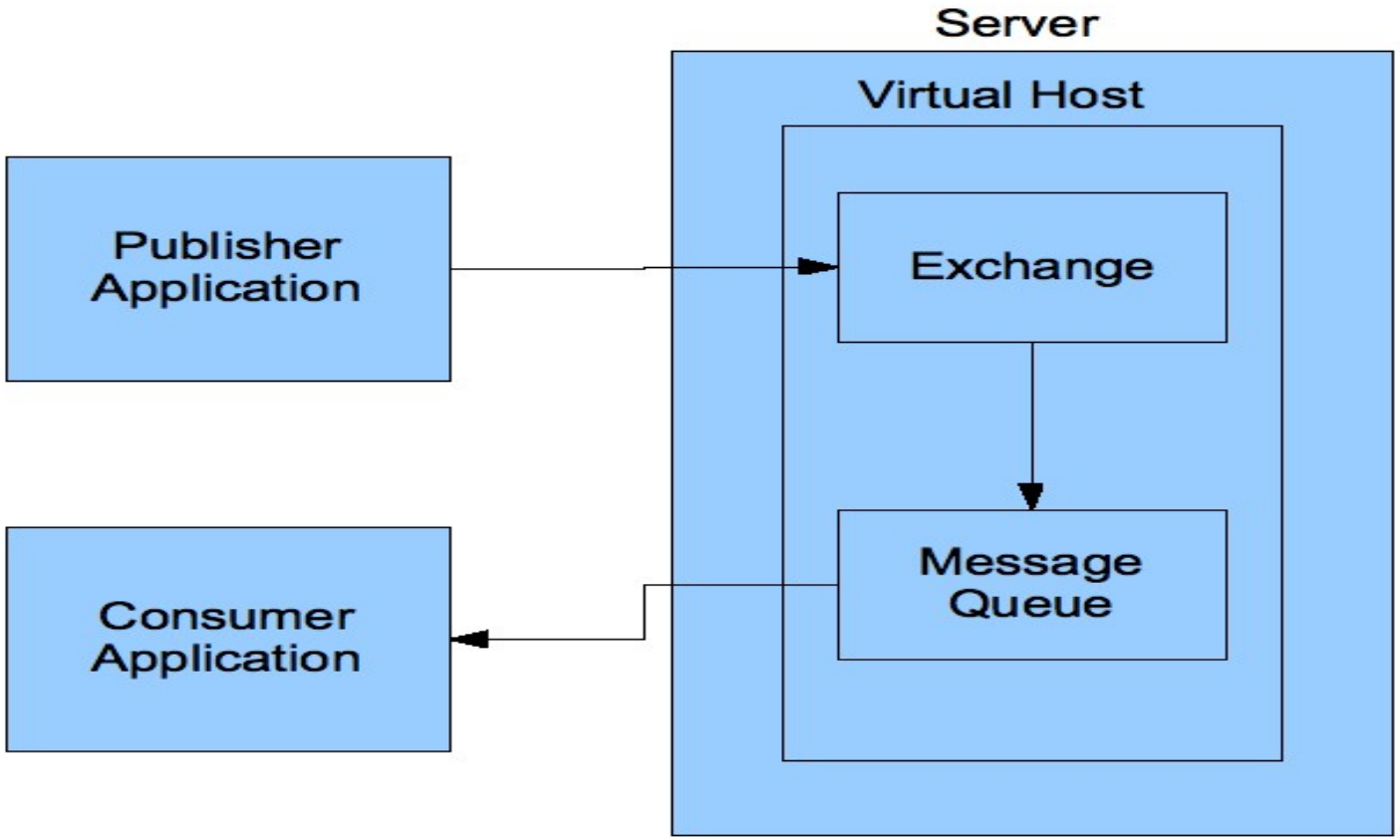
AMQP Protocol

- Network wire-level protocol
  - Defines how clients and brokers talk
  - Data serialization (framing), heartbeat
  - Hidden inside client libraries
- AMQP Model
  - Defines routing and storing of messages
  - Defines rules how these are wired together
  - Exported API

AMQP

- Information is organized into “frames”
- Independent threads of control within a single socket connection are called “channels”
- For each channel, frames run in sequence
- Each frame consists of header (type, channel id, payload size), payload, frame end packet
- Frames can be protocol methods (commands), structured content (message headers), data

AMQP model


Server
Virtual Host
Publisher Application
Exchange
Message Queue
Consumer Application

AMQP Exchange

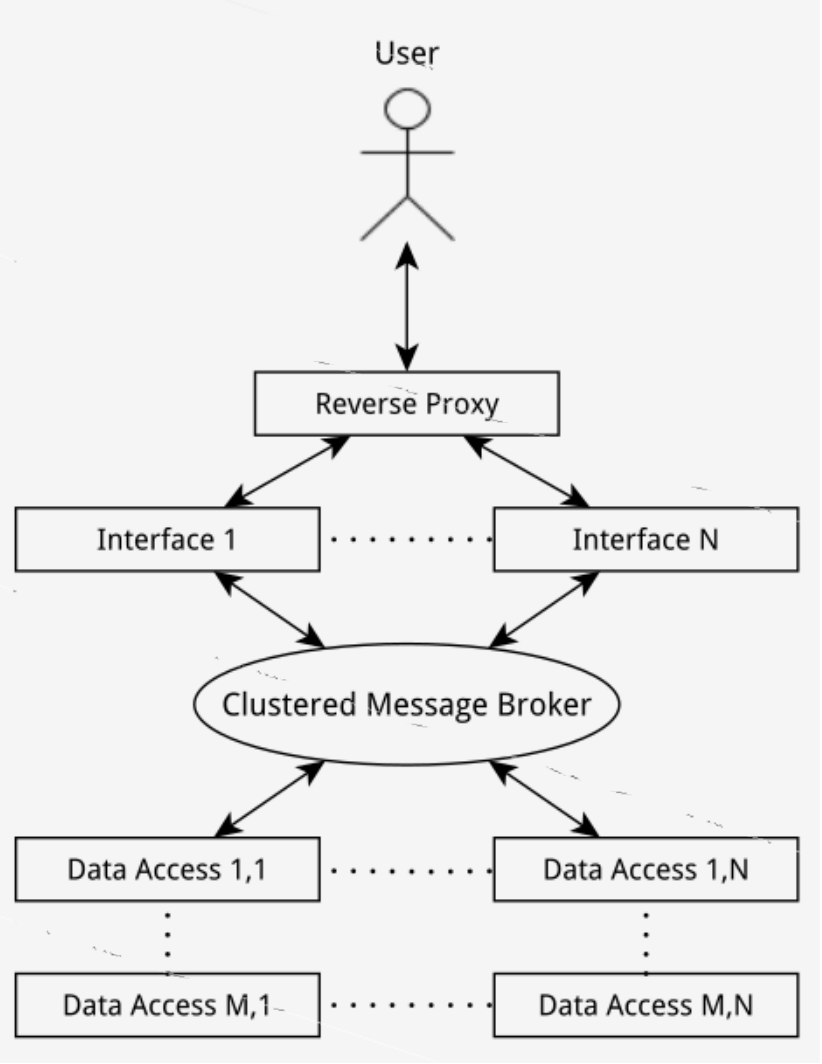
- A message routing agent
- Can be durable – lasts till explicitly deleted
- Can be temporary – lasts till server shuts down
- Can be auto-deleted – lasts till no longer used
- There are several types of exchanges, each implements a particular algorithm
- Each message is delivered to each qualifying queue
- “Binding” - a link between queue and exchange

RabbitMQ

- RabbitMQ is a broker written in Erlang
- RabbitMQ team also provides Java and .NET clients
- Implements AMQP 0-8 spec
- Experimental products include AMQP-over- HTTP + Javascript libraries, Erlang client, gateway for STOMP clients, XMPP (Jabber) gateway
- Supports clustering for High Availability and scalability
  - Implemented with Erlang distributed nodes
  - Data/state replication with full ACID properties
  - Exception: queues (visible/reachable from everywhere, reside on node which created them – to be changed in future)
  - 1 client = 1 socket. Cluster helps scale! High Availability

Example: Scalable, HA application for IVOA access

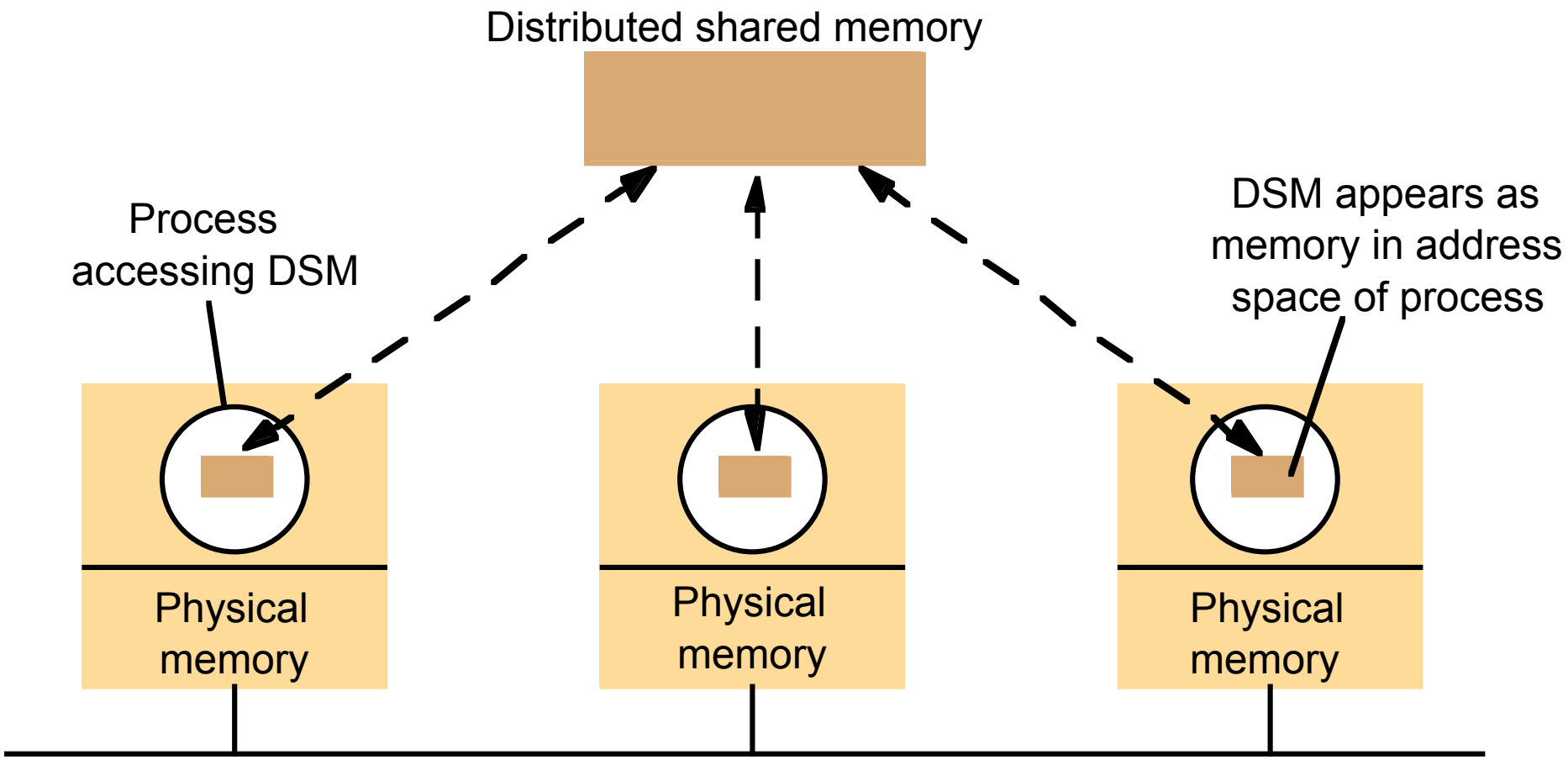
- System for accessing IVOA data developed at INAF - OAT
- Reverse proxy implements load balancing (Java on Glassfish)
- Message broker: RabbitMQ
- Data Access: existing applications



Distributed shared memory (DSM) & tuple spaces

- DSM is an abstraction used for sharing (unstructured) data between computers not sharing physical memory:
  - processes access DSM like ordinary memory within their address space;
  - the underlying middleware ensures that processes executing at different nodes may observe the updates.
- Tuple spaces offer a higher-level perspective:
  - they are associative spaces, giving processes a form of content-addressable memory;
  - data is semi-structured.

Distributed Shared Memory (DSM)



Distributed Shared Memory (DSM)

- W.r.t. message passing, DSM allows processes to **directly** access shared items:
  - hence, it is not appropriate in client/server systems;
  - a form of message passing is still needed in distributed systems: the DSM runtime must send updates in messages between computers;
  - each computer has a **local copy** of recently accessed data items (for speed of access).
- Implementation issues: **replication** and **caching** management.
- Parallel architectures: NUMA systems (up to 64 processors).

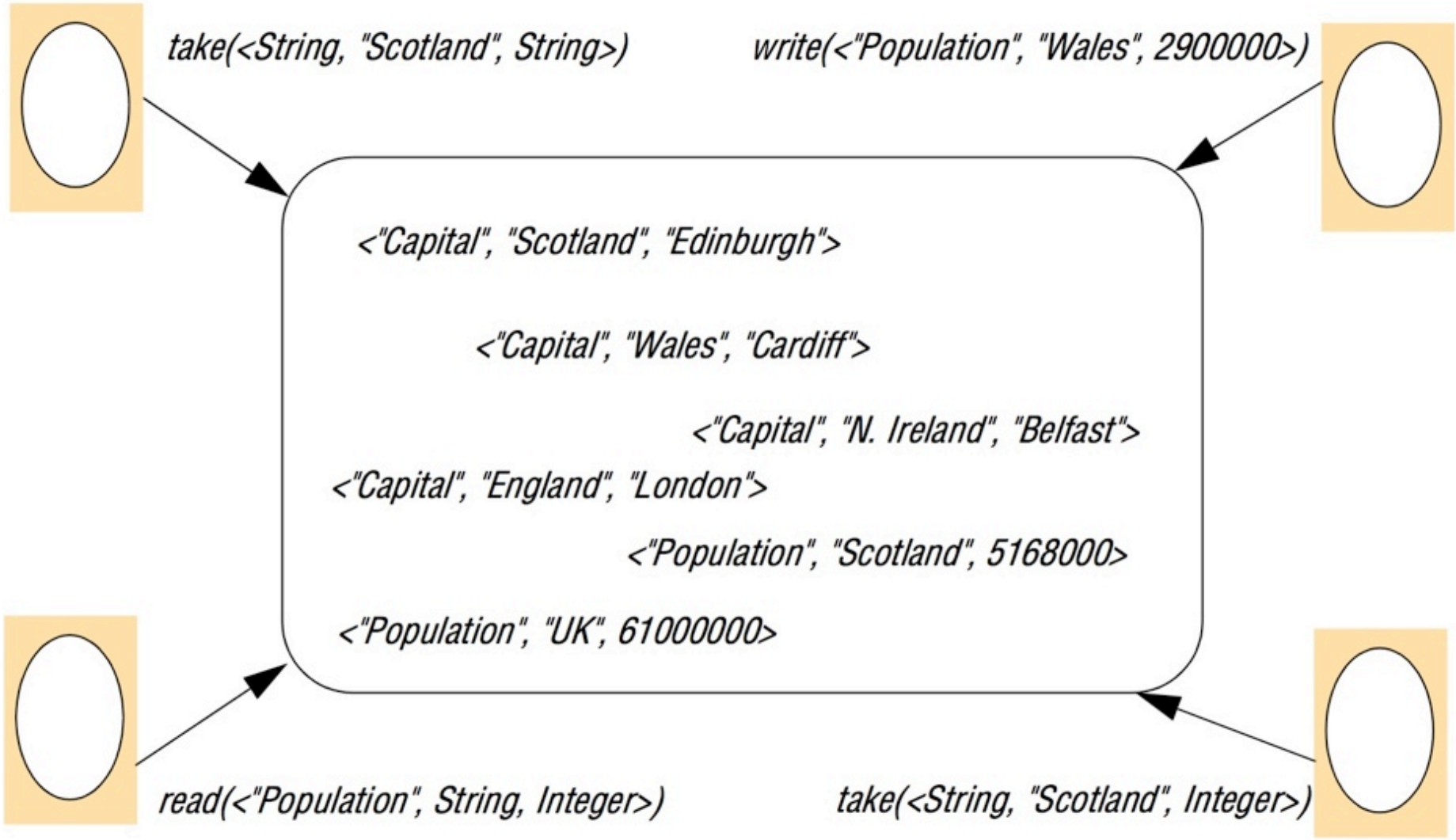
Distributed Shared Memory (DSM) vs. Message passing

- Entailing the use of **asynchronous** communication, **DSM** is comparable with **message passing**, rather than with **request-reply** communication.
- **Service** offered:
  - DSM: variables are **accessed directly** (processes may **fail** due to erroneous modifications);
  - Message passing: variables are **marshalled**, **transmitted** and **unmarshalled** (processes are **protected** from one another).
- **Efficiency**: there is no definite answer as to whether DSM performs better than message passing: it depends on the specific use cases.
- **Time uncoupling**: DSM can be made persistent, whence processes may execute with non-overlapping lifetimes.

Tuple Spaces

- introduced by Gelernter (1985) as a new form of distributed computing based on **generative communication**.
- Processes communicate by **placing tuples in a common space**, where other processes can **read** or **remove** them.
- Linda programming model (1992):
  - C-Linda, TCP-Linda, LinuxTuples (C);
  - CppLinda (C++);
  - JavaSpaces, IBM's TSpaces (Java);
  - Erlinda (Erlang);
  - ...

The tuple space abstraction



- In order to allow processes to **synchronize**, both the read and take operations **block** until there is a **matching** tuple in the tuple space.
- As an example, let us implement a **shared semaphore** in a tuple space:
  - up(S) = write("semaphore", S);
  - down(S) = take("semaphore", S);

Distributed coordination

- Pros: **flexible** and **general** model.
- Cons: **difficult** to **implement** in a really distributed way
  - Original design: a centralized server
  - Later implementations: p2p, DHT, replicas
- **Distributed sharing**: both **space** and **time uncoupling** are granted.

Variations on Linda programming model

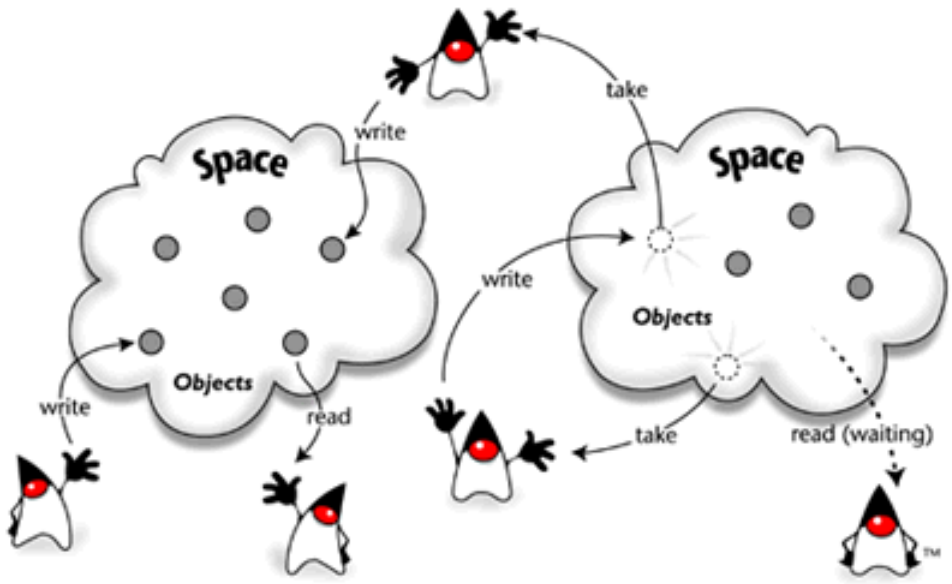
- In the original Linda model there was only a single tuple space, but this is not optimal as the number of tuples increases:
  - unintended **aliasing** of tuples;
  - modern systems allow **multiple tuple spaces** with the ability to create new spaces dynamically.
- **Centralized** vs. **distributed** implementations: the latter provide more **fault tolerance**.
- Modelling everything as **unordered sets** (tuples become a special case).
- Moving from tuples of (typed) data items to objects, yielding **object spaces** (e.g., JavaSpaces).

Implementation issues about replication

- State machine approach [Bakken and Schlichting, 1995]:
  - a tuple space behaves like a **state machine**;
  - state is changed in response to **events** received from other replicas or from the environment;
  - replicas must **start** in the **same state** (empty space);
  - replicas must **execute** events in the **same order** (totally ordered multicast);
  - replicas must react **deterministically** to each event.

Introduction to JavaSpaces (http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/)

- JavaSpaces is a high-level tool for building **distributed** and **collaborative** applications.
- It is based on the concept of shared network-based space that serves as
  - object storage;
  - exchange area.
- JavaSpaces allows the programmer to “merge” process coordination and **data sharing**.
- It provides a sort of **persistent** distributed memory.



JavaSpaces Operations

- Processes use the **persistent storage** of a JavaSpaces to store objects and to communicate.
- Processes interact with a space through the following set of operations:
  - **`write()`**: writes new objects into a space;
  - **`take()/takeIfExists()`**: retrieves objects from a space according to a given template;
  - **`read()/readIfExists()`**: makes a local copy of objects in a space according to a given template;
  - **`notify()`**: notifies a specified object when entries that match the given template are written into a space.

Spaces and Entries

- All operations are based on an object implementing the **net.jini.space.JavaSpace** interface.
- A space stores **entries**: collections of typed objects that implements the **Entry interface**.

```
import net.jini.core.entry.*;

public class MessageEntry implements Entry {
    public String content;

    public MessageEntry() {
    }

    public MessageEntry(String content) {
      this.content = content;
    }

    public String toString() {
      return "MessageContent: " + content;
    }
}
```

Writing and Reading an Entry

```
JavaSpace space = getSpace();
MessageEntry msg = new MessageEntry();
msg.content = "Hello there";
space.write(msg, null, 60 * 60 * 1000);
```

- null field → no Transaction object is managing the operation.
- 60 * 60 * 1000 → lease of one hour for the entry.

```
MessageEntry template = new MessageEntry();
MessageEntry output = (MessageEntry) space.read(template, null,
                                    Long.MAX_VALUE);
```

- null field → template will match any entry.
- Long.MAX_VALUE → timeout for the matching read.

Writing Client

```
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;
import net.jini.lookup.*;

public class WritingClient {
    public static void main(String argv[]) {
          try {
             MessageEntry msg = new MessageEntry();
             msg.content = "Hello there";
             System.out.println("Searching for a JavaSpace...");
             Lookup finder = new Lookup(JavaSpace.class);
             JavaSpace space = (JavaSpace) finder.getService();
             System.out.println("A JavaSpace has been discovered.");
             System.out.println("Writing a message into the space...");
             space.write(msg, null, 60*60*1000);
          } catch(Exception e) {
             e.printStackTrace();
          }
       }
}
```

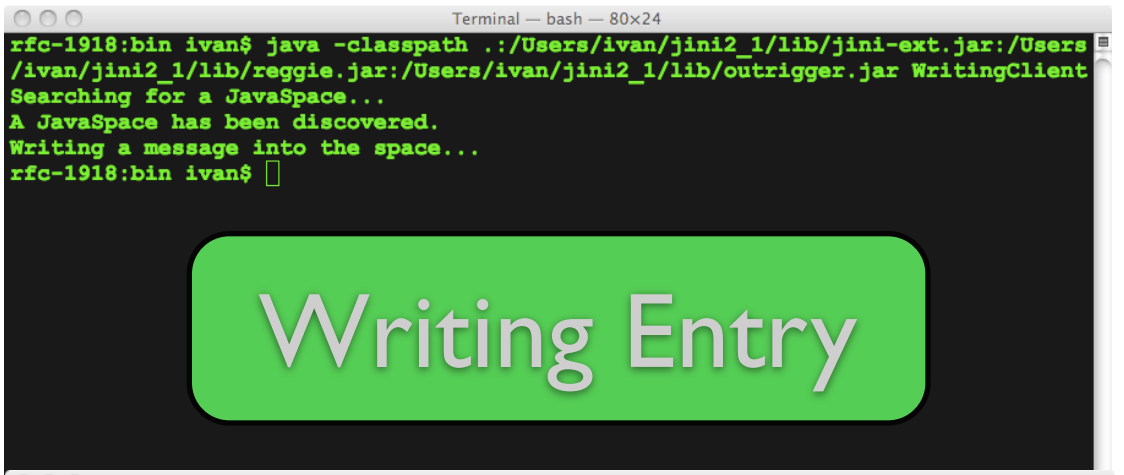
Reading Client

```
import net.jini.space.JavaSpace;

public class ReadingClient {
   public static void main(String argv[]) {
      try {
         MessageEntry msg = new MessageEntry();
         msg.content = "Hello there";
         System.out.println("Searching for a JavaSpace...");
         Lookup finder = new Lookup(JavaSpace.class);
         JavaSpace space = (JavaSpace) finder.getService();
         System.out.println("A JavaSpace has been discovered.");
         MessageEntry template = new MessageEntry();
         System.out.println("Reading a message from the space...");
         MessageEntry result = (MessageEntry) space.readIfExists(template,
                                                    null, 5000);
         System.out.println("The message read is: "+result.content);
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Sample Run...

JavaSpaces is running



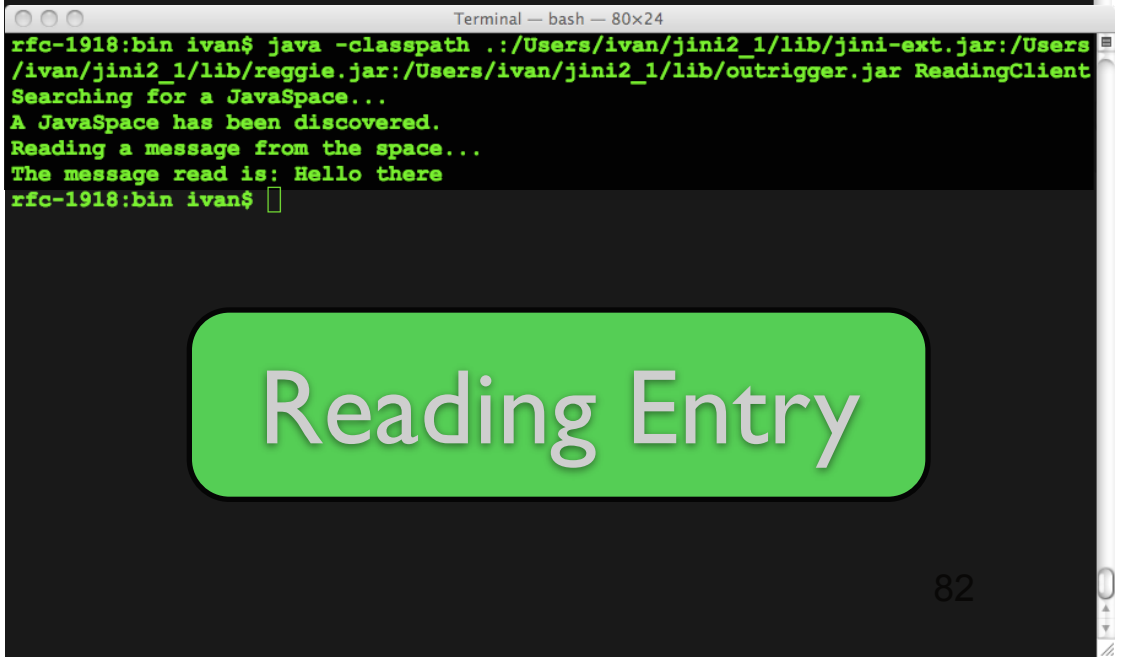


Figure 6.27
Summary of indirect communication styles

	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