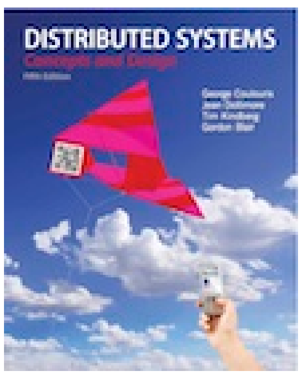
Slides for Chapter 6: Indirect Communication



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

Edition 5, © Addison-Wesley 2012

Text extensions to slides © David E. Bakken, 2012-2020

Introduction [6.1]

- Cambridge researchers:
  - "All problems in computer science can be solved by another level of indirection."
- Jim Gray (RIP)
  - "There is no performance problem that cannot be solved by eliminating a level of indirection."
- **Indirect communication**: communication between entities in a DS through an intermediary with no direct coupling between sender and receiver(s).
- Lots of variations in
  - Intermediary
  - Coupling
  - Implementation details and tradeoffs therein

Indirect communication (cont.)

- Why have decoupled comms? Client-server interaction
  - Hard to change server to one with same functionality
  - Harder to deal with failure
  - …. Other change is expected (what kinds?)
- Note: continuum between server “group” and intermediary..
  - We look at group communication in Sec 6.2

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

Q: is time/space uncoupling same as asynchronous invocation?

© Pearson Education 2012

Group communication [6.2]

- **Group communication**: Send messages to a group endpoint
  - Delivered to all members (modulo reliability guarantees)
  - Sender not aware of identity of receivers
  - Ergo, (thin) abstraction layer above IP multicast or an overlay net
- Adds a lot of value
  - Detecting failures
  - Managing group membership (processes in the group)
  - Reliability guarantees
  - Ordering guarantees

Group communication (cont.)

- Very useful building block for DSs, esp. reliable ones
  - Reliable dissemination of info to large # “clients” (esp. finance)
  - Collaborative applications: multiple users with common view
  - Wide range of fault-tolerance building blocks
    - Consistent update of replicated data
    - Highly available (replicated) servers
- More on group communications next:
  - Programming models
  - Implementation issues
  - Case study: JGroups toolkit [NOT TESTABLE]

Programming model [6.3.1]

- Central abstraction: group & associated membership
  - Processes join (explicitly) or leave (explicitly or by failure)
  - Send single message to the group of N, not N unicast messages
- Compare and contrast with IP multicast?
- Early work started in the late 1980s, still going strong

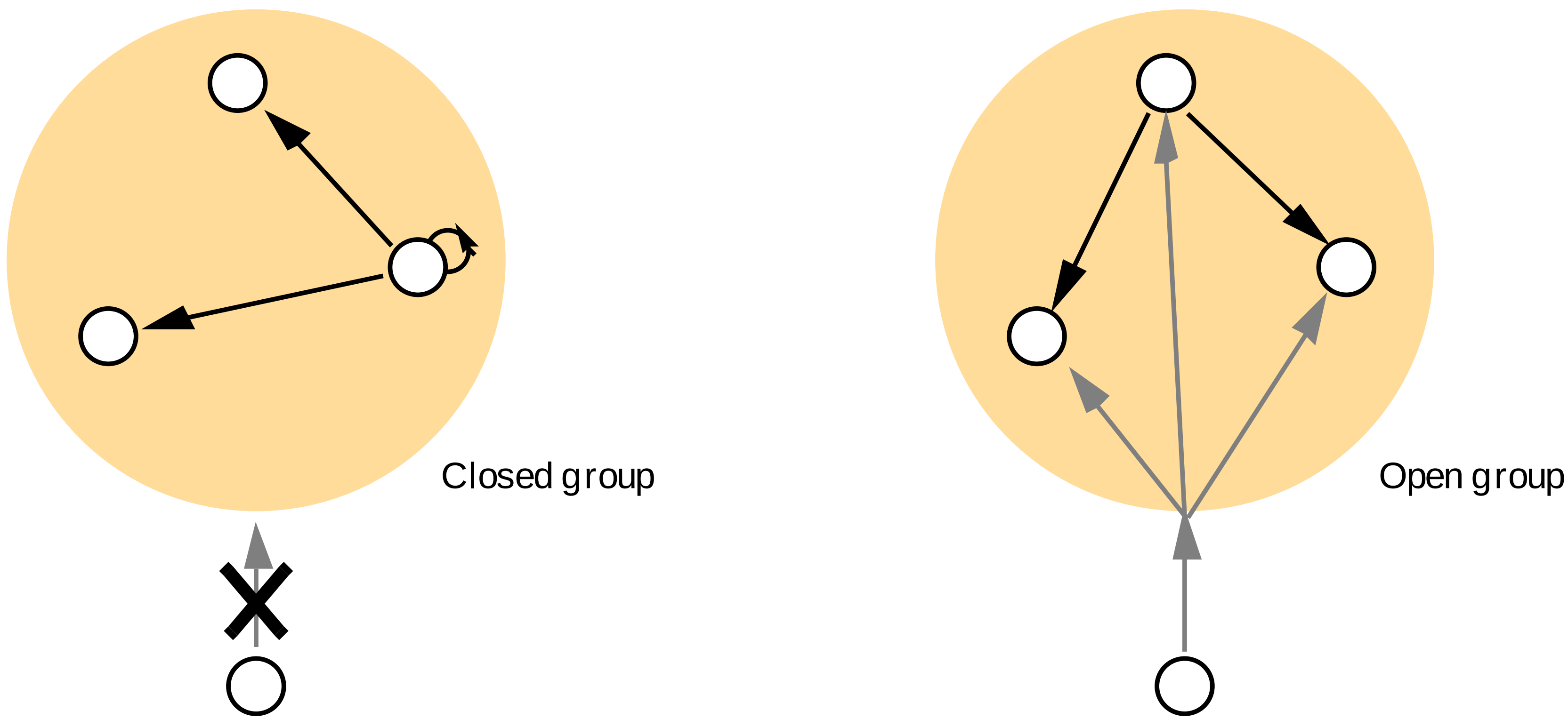
Process groups and object groups

- Most research on **process groups**
  - Abstraction: resilient process
  - Messages delivered to a process endpoint, no higher
  - Messages typically unstructured byte arrays, no marshalling etc
  - Level of service ≈ socket
- **Object group**: higher level approach
  - Collection of objects (same class!) process same invocations
  - Replication can be transparent to clients
    - Invoke on single object (proxy)
    - Requests sent by group communication
    - Voting in proxy usually
  - Research started in mid 1990s (Electra, Eternal, AQuA)
- Process groups still more widely researched & deployed

Other key distinctions in group comm. services

- **Closed group**: only members may multicast to it
  - Useful: coordinating among cooperating servers (usually replicas)
- **Open group**: a process outside group may send to it
  - Useful: delivering events to interested parties, client request to server replica group
- **Overlapping groups**: entities may belong to >1 group
- **Non-overlapping groups**: 0 or 1 groups for an entity
- Synchronous and asynchronous systems
- Note: above has HUGE impact on multicast algorithms
  - Big reason why lots of research on this!
  - …. And that is even without Byzantine failure

Figure 6.2
Open and closed groups



© Pearson Education 2005

Implementation issues [6.2.2]

- Reliable delivery
  - Unicast delivery reliability properties (note: not my favorite terms!)
    - **Delivery integrity**: message received same as sent, never delivered twice
    - **Delivery validity**: outgoing message eventually delivered
  - Group communication reliability properties build on this
    - **Delivery integrity**: deliver message correctly at most once to group members
      - Note: stronger than RPC delivery guarantees!
    - **Delivery validity**: message sent will be eventually delivered (if not all group members fail)
    - **Agreement/consensus**: Delivered to all or none of the group members
      - Note: also called **atomic delivery**

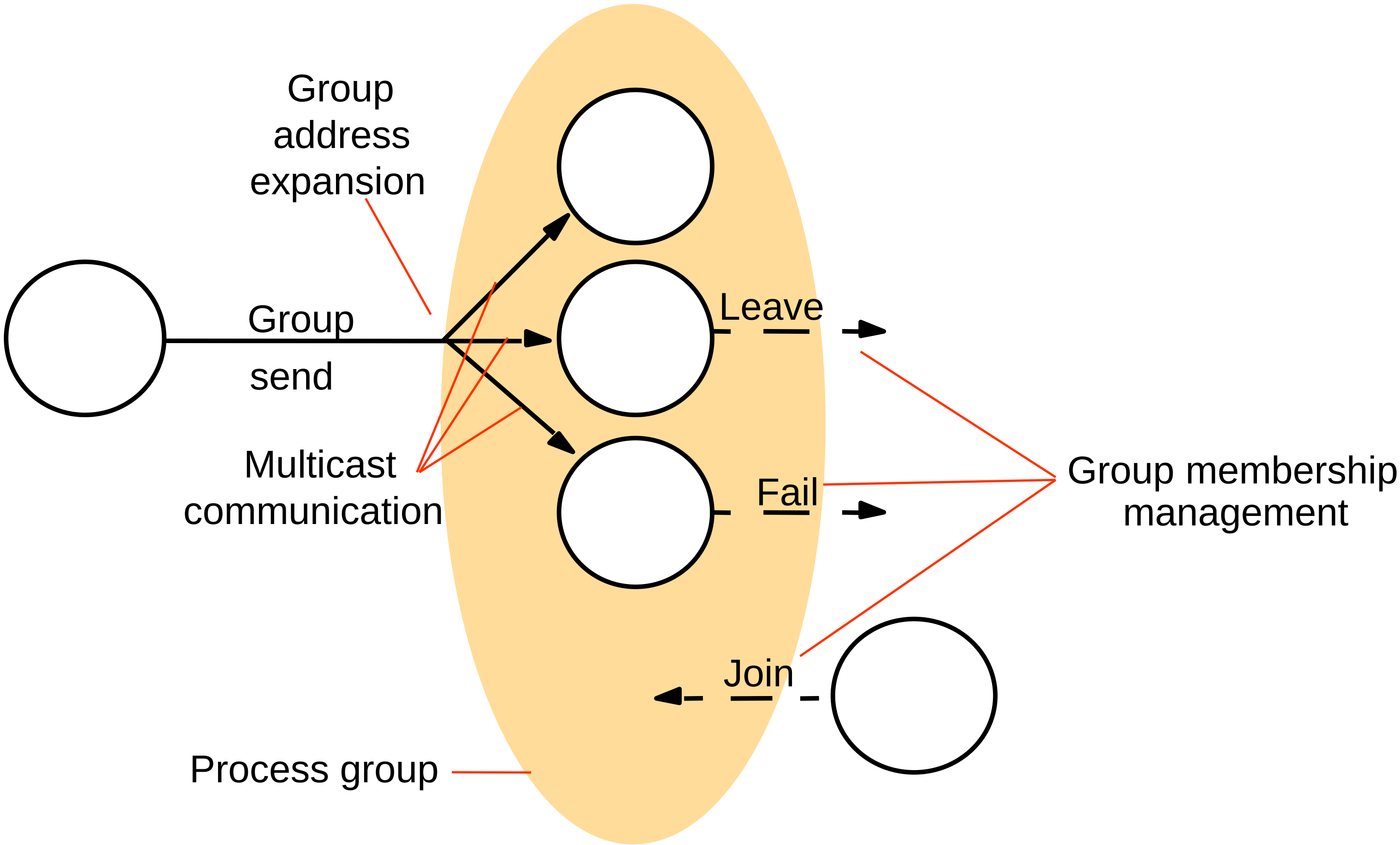
Ordered delivery

- Possible strengths of ordering
  - **Fifo ordering**: first-in-first-out from a single sender to the group
  - **Causal ordering**: preserves potential causality, happens before (Chap 14)
  - **Total ordering**: messages delivered in same order to all processes
- Perspective (not testable unless later covered…)
  - Strong reliability and ordering is expensive: scale limited
  - More probabilistic approaches & weaker delivery guarantees researched a lot last decade

Group membership management

- Key elements
  - Provide interface for group membership changes
  - Failure detection
  - Notifying members of group membership changes
    - Sometimes with strong properties: virtual synchrony
  - Performing group address expansion
  - Q: what of these does IP multicast perform?

Figure 6.3
The role of group membership management

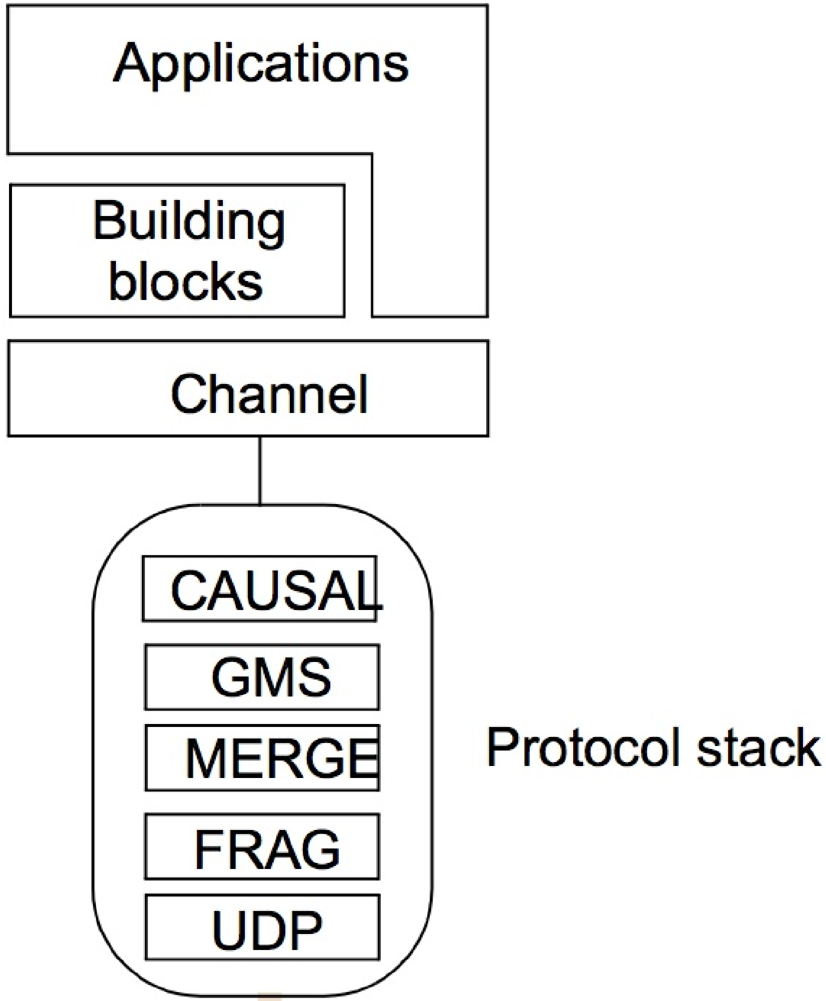


© Pearson Education 2012

Case study: JGroups toolkit [NOT TESTABLE]

- Java toolkit, based on Cornell/Birman's research
- Architecture
  - Channel: most primitive API
  - Building blocks: higher-level APIs built on top of channels
  - Protocol stack: different underlying comms. protocols

Figure 6.4
The architecture of JGroups



© Pearson Education 2012

JGroups channels

- **Channel object**: handle/reference for a group
  - Note: different from channel-based publish-subscribe (6.3.1)
- Sends messages with some form of reliable multicast
- Basic operations
  - `connect` to a named group
  - Leave a group: `disconnect` operation
  - `close`: shut down channel object
- Other operations (admin stuff)
  - `getView` returns current member list
  - `getState` returns app state history

JGroups example

- Simple example: intelligent fire alarm sends "Fire!" message to group
- To raise the alarm:

  FireAlarmJG alarm = ***new*** *FireAlarmJG();*

  Alarm.raise();

- To receive the alarm:

  FireAlarmConsumerJG alarmCall = ***new*** *FireAlarmConsumerJG();*

  String *msg = alarmCall.await();*

  System.out.println("Alarm received: " + msg);

Figure 6.5
Java class *FireAlarmJG*

```
======================================
import org.jgroups.JChannel;
public class FireAlarmJG {
public void raise() { // raise alarm, i.e. send “Fire!” message
    try {
        JChannel channel = new JChannel();
        channel.connect("AlarmChannel");  // can create group
        Message msg = new Message(null, null, "Fire!");
        channel.send(msg);
    }
    catch(Exception e) {
    }
}
```


© Pearson Education 2012

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;
     }
  }
}
```


© Pearson Education 2012

JGroups building blocks & protocol stack

- Building blocks examples
  - `MessageDispatcher`: sends msg, waits for (some) replies
  - `RpcDispatcher`: invokes a method on all objects, wait for replies
  - `NotificationBus`: distribited event bus, with any serializable Java object
- Protocol stack (some, from Fig 6.4):
  - UDP: obvious, but uses IP multicast with UDP
  - FRAG: message fragmentation and reassembly
  - MERGE: deals with network partitioning (multiple versions)
  - GMS: group membership
  - CAUSAL: causal ordering
  - (lots of other protocols available: FIFO, total, discover, failure detection, encryption, flow-control, … & layers stack in any order)

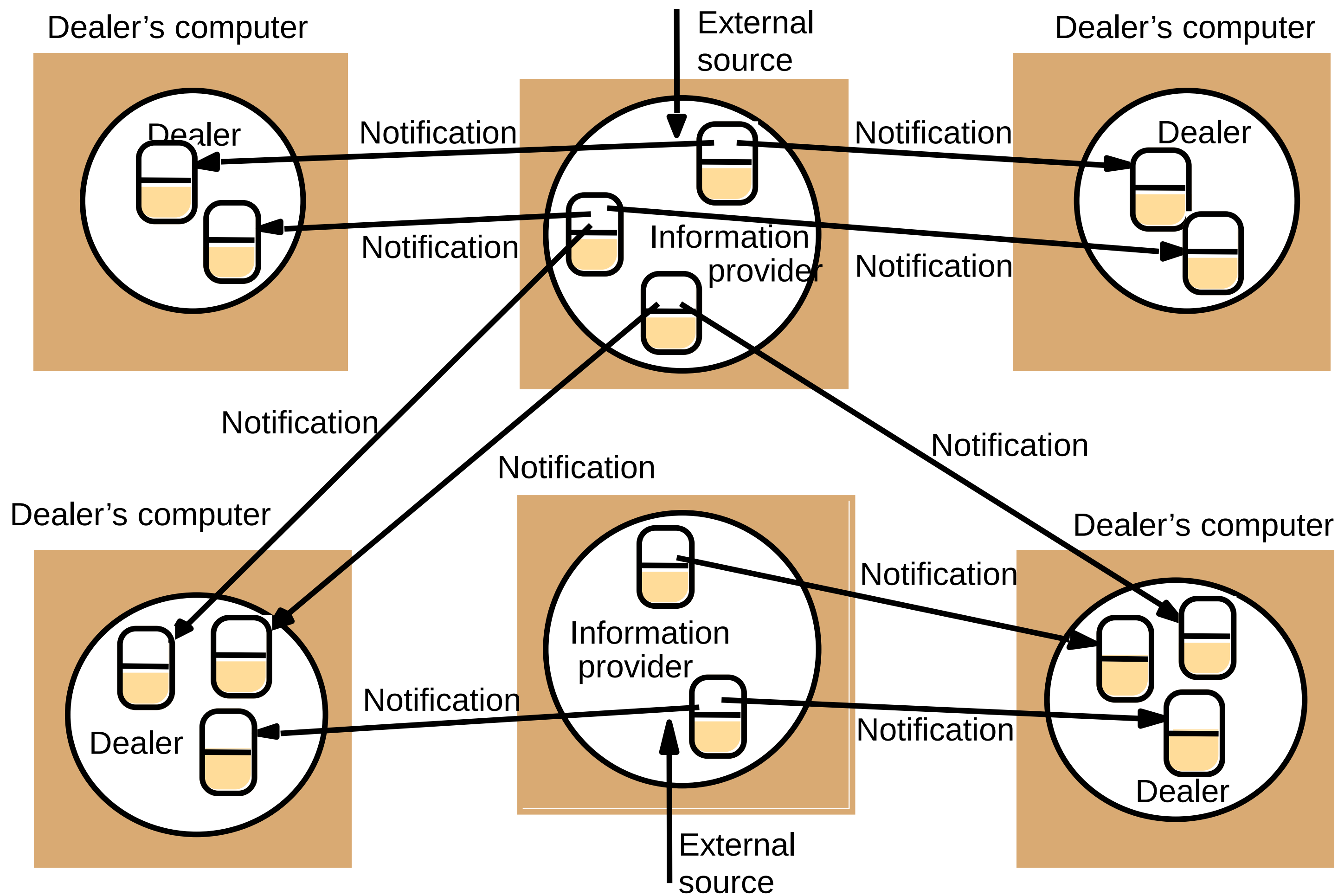
Public-subscribe systems [6.3]

- 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 (chap 21)

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



© Pearson Education 2005

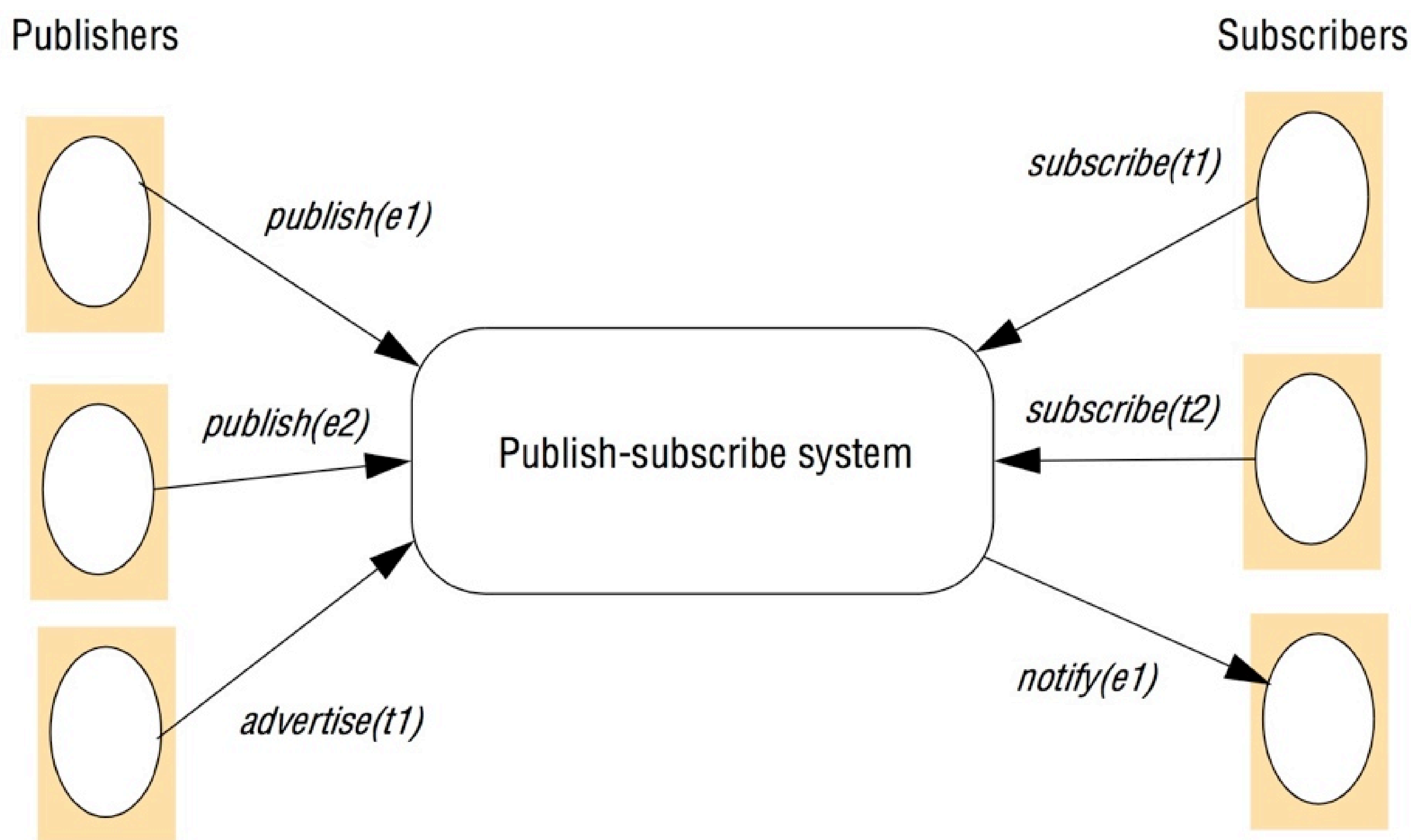
Characteristics of pub-sub systems

- 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!
- Possible delivery guarantees
  - 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)`
  - 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(f)`
  - Cancel their subscription: `unsubscribe(f)`

Figure 6.8
The publish-subscribe paradigm



© Pearson Education 2005

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 (DDS precursor)
- **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
  - Ranges from 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.)

- Other kinds
- **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
  - For more, see ACM's Distributed Event-Based Systems (DEBS) conference

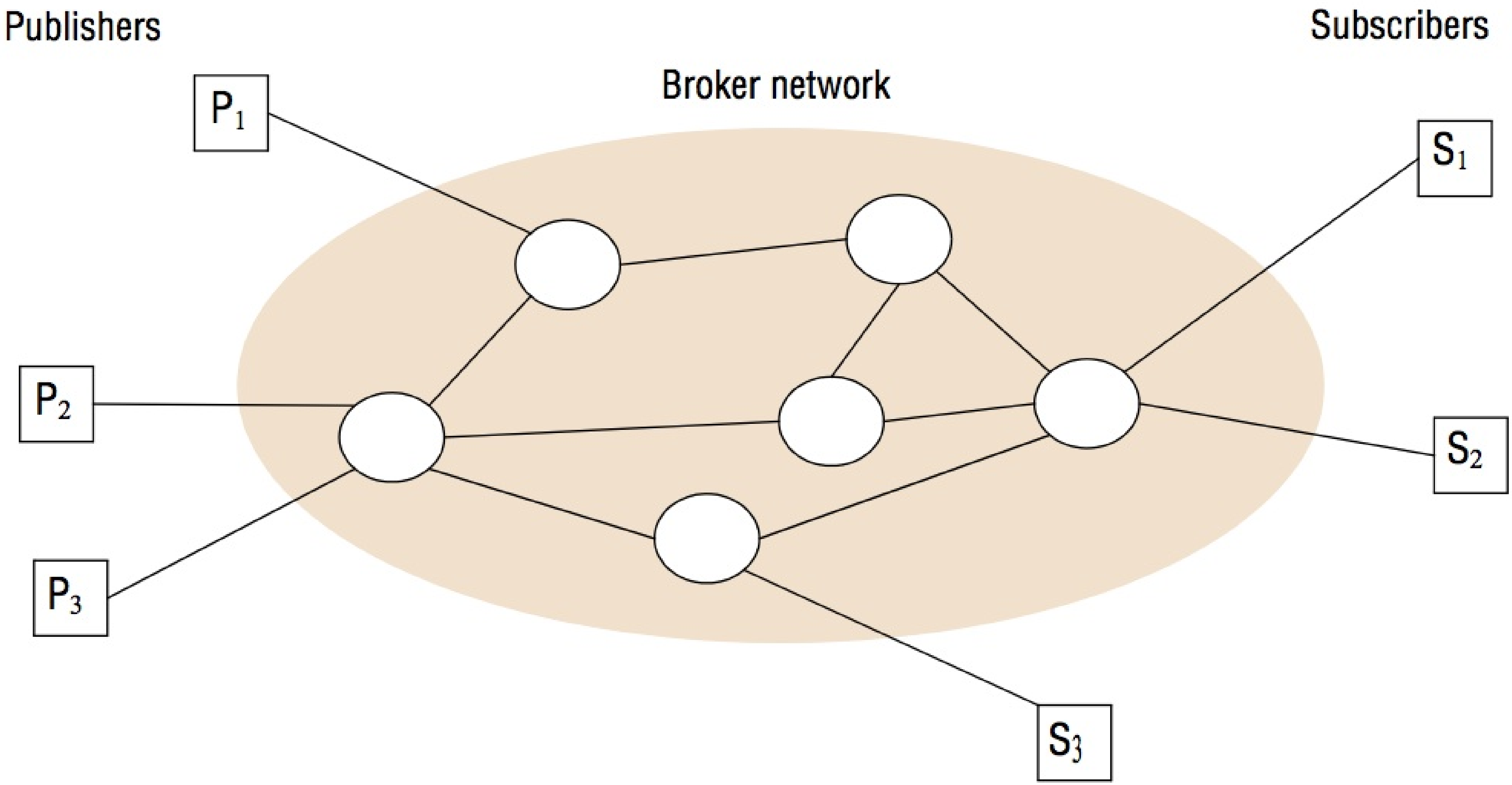
Implementation issues [6.2.3]

- 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
- Q: Limitations?
- Most implementations are **network of brokers (Fig 6.9)**
  - E.g., GridStat
- 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

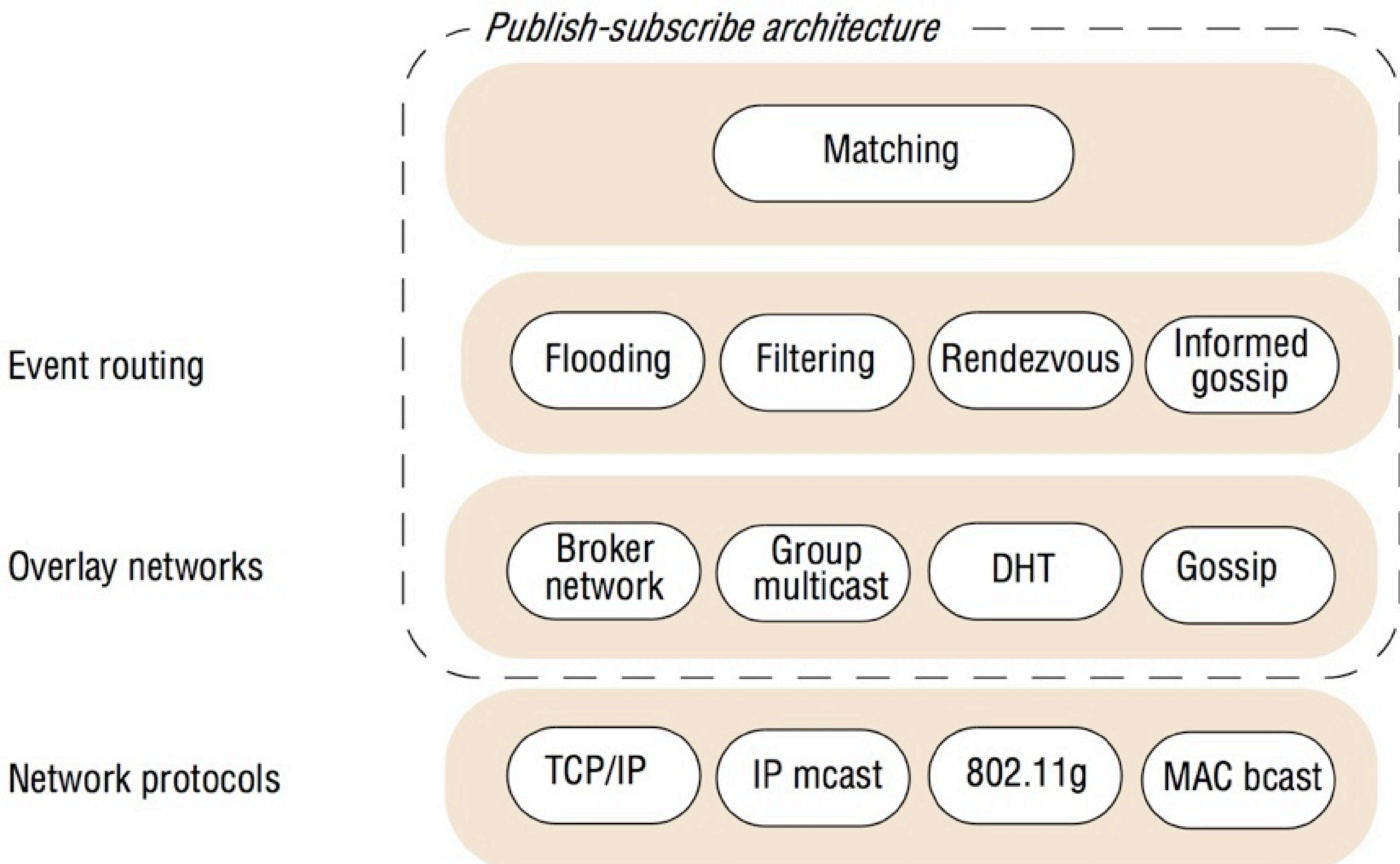


© Pearson Education 2005

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



© Pearson Education 2005

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
  - Detail:
    - 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
    matchlist := match(e, subscriptions)
    send notify(e) to matchlist;
    fwdlist := match(e, routing);
    send publish(e) to fwdlist - x;

upon receive subscribe(subscription s) from node x
    if x is client then
        add x to subscriptions;
    else add(x, s) to routing;
    send subscribe(s) to neighbours - x;
```


© Pearson Education 2005

Implementation choices in CBR (cont.)

- **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;
```


© Pearson Education 2005

Figure 6.13
Example publish-subscribe system

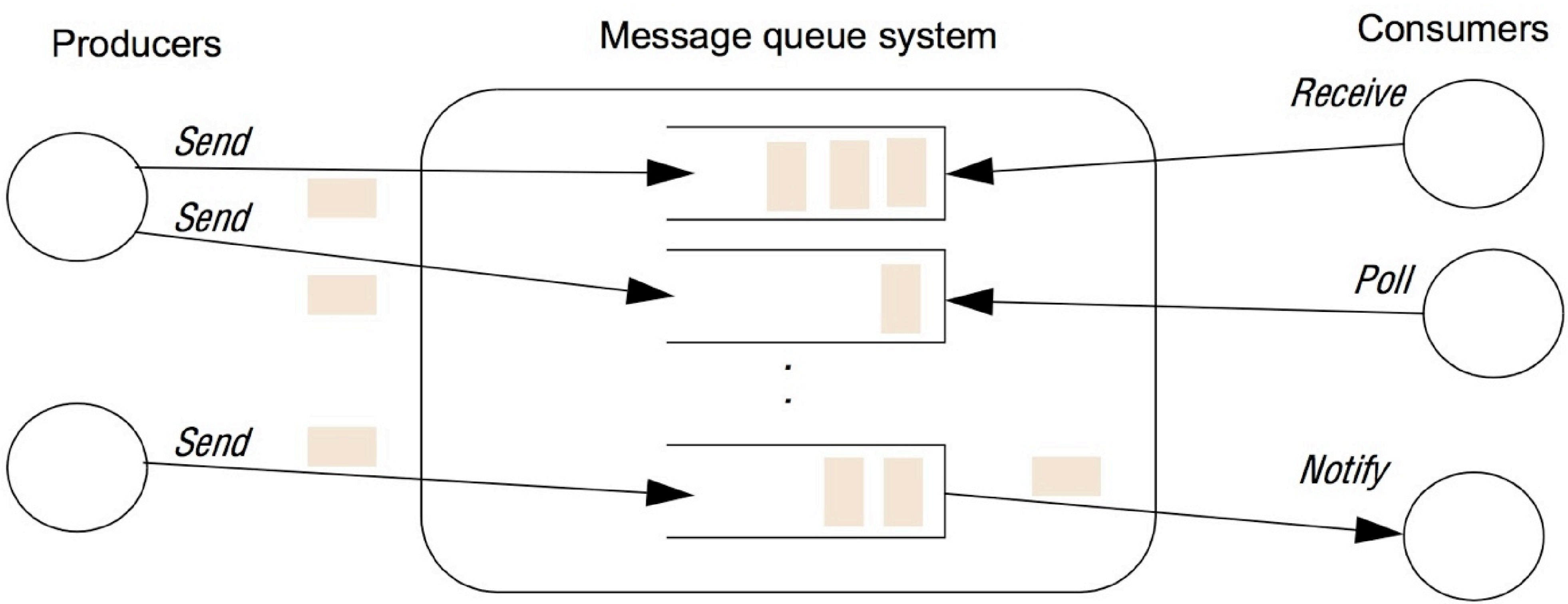
System (and further reading)	*Subscription model*	*Distribution model*	*Event routing*
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvouz [Oki *et al.* 1993]	Topic-based	Distributed	Ffiltering
Scribe [Castro *et al.* 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni *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

© Pearson Education 2005

Message queues [6.4]

- (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
  - Main 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



© Pearson Education 2005

Programming model [6.4.1] (cont.)

- 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 AQ
  - Messages are a row in a (relational) database
  - Queues are database tables that can be SQL-queried against

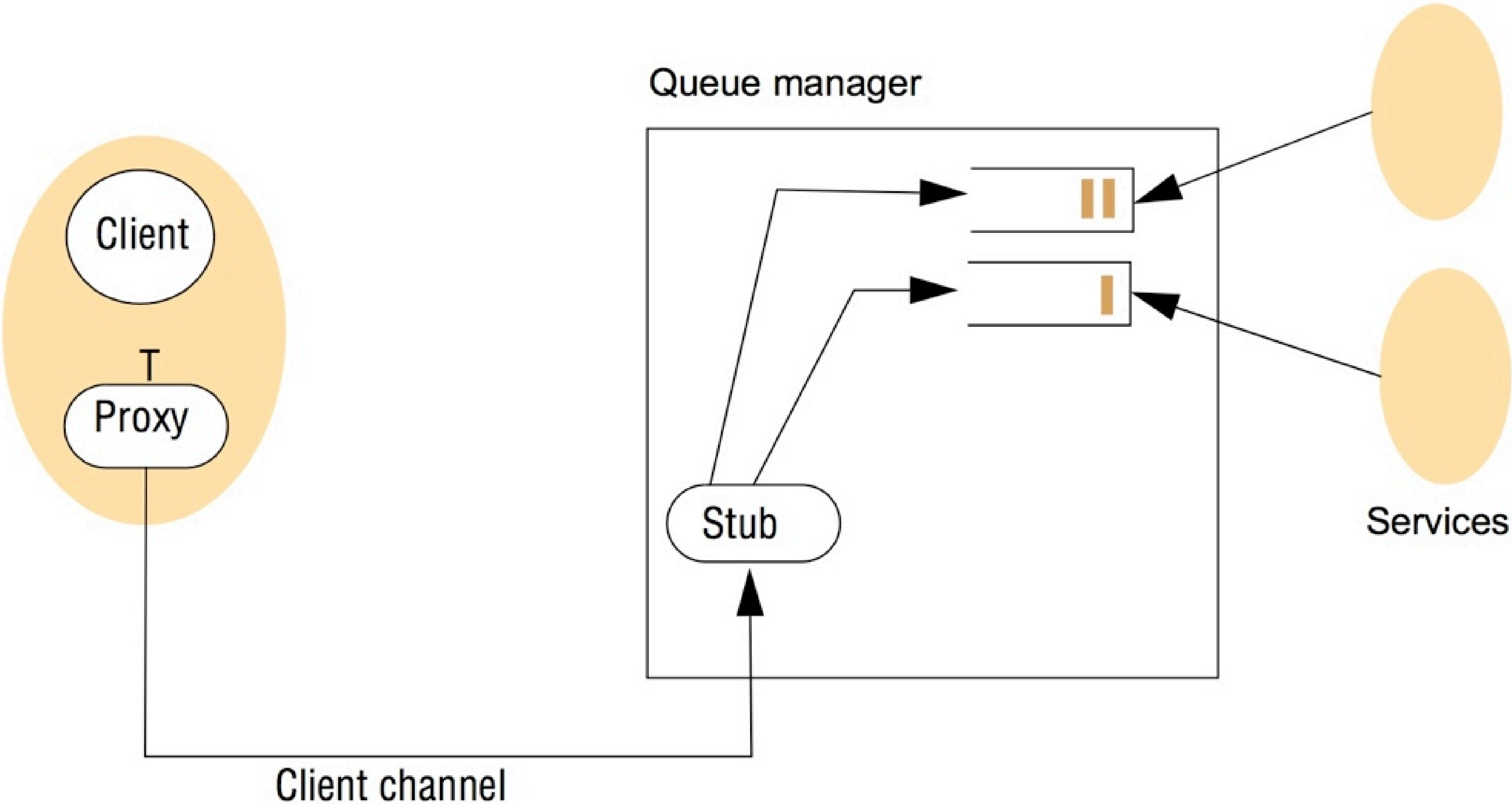
Programming model (cont)

- 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)
- Q: How different from message passing from Chap 4?

Implementation issues [6.3.2]

- Key choice: centralized vs. distributed implementation
  - Tradeoffs?
- Case study: IBM Websphere MQ
  - **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) vi a **client channel** (w/proxy+stub)

Figure 6.15
A simple networked topology in WebSphere MQ



© Pearson Education 2005

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
- 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

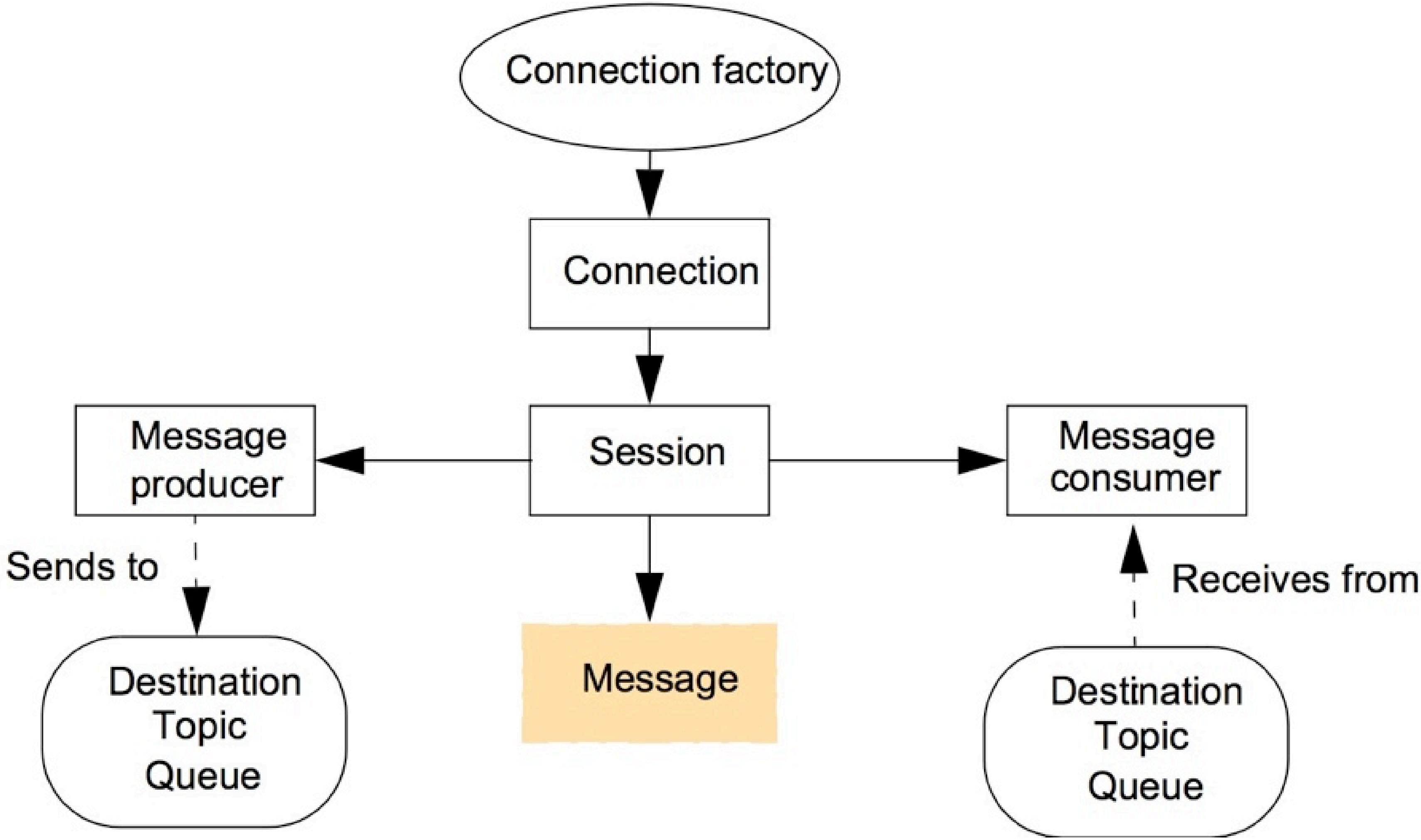
Case study: Java Messaging Service (JMS) [6.4.3] [NOT TESTABLE]

- 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)

Programming with JMS

- First create a **connection** from client to providor with **connection factory**
  - `TopicConnection` or `QueueConnection`
- Use connection to create ≥1 **session**
  - Series of ops 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



© Pearson Education 2005

JMS session objects

- Message has 3 parts
  - Header: everything needed to identify & route msg
    - Destination, priroity, 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
    1. Block with receive operation
    2. Create message listener object with a **callback object** `onMessage`

Figure 6.17
Java class *FireAlarmJMS*

```
// Usage: alarm.raise()
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS { // more complex than Jgroups: create connection, session, publisher, message
// Lines 2-5 find the right connection factory and topic with JNDI (Lines 2-5)
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, // false means not transactional 8
                                        Session.AUTO_ACKNOWLEDGE);   // session ACKS msg receipt 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
}
```

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

Figure 6.18
Java class *FireAlarmConsumerJMS*

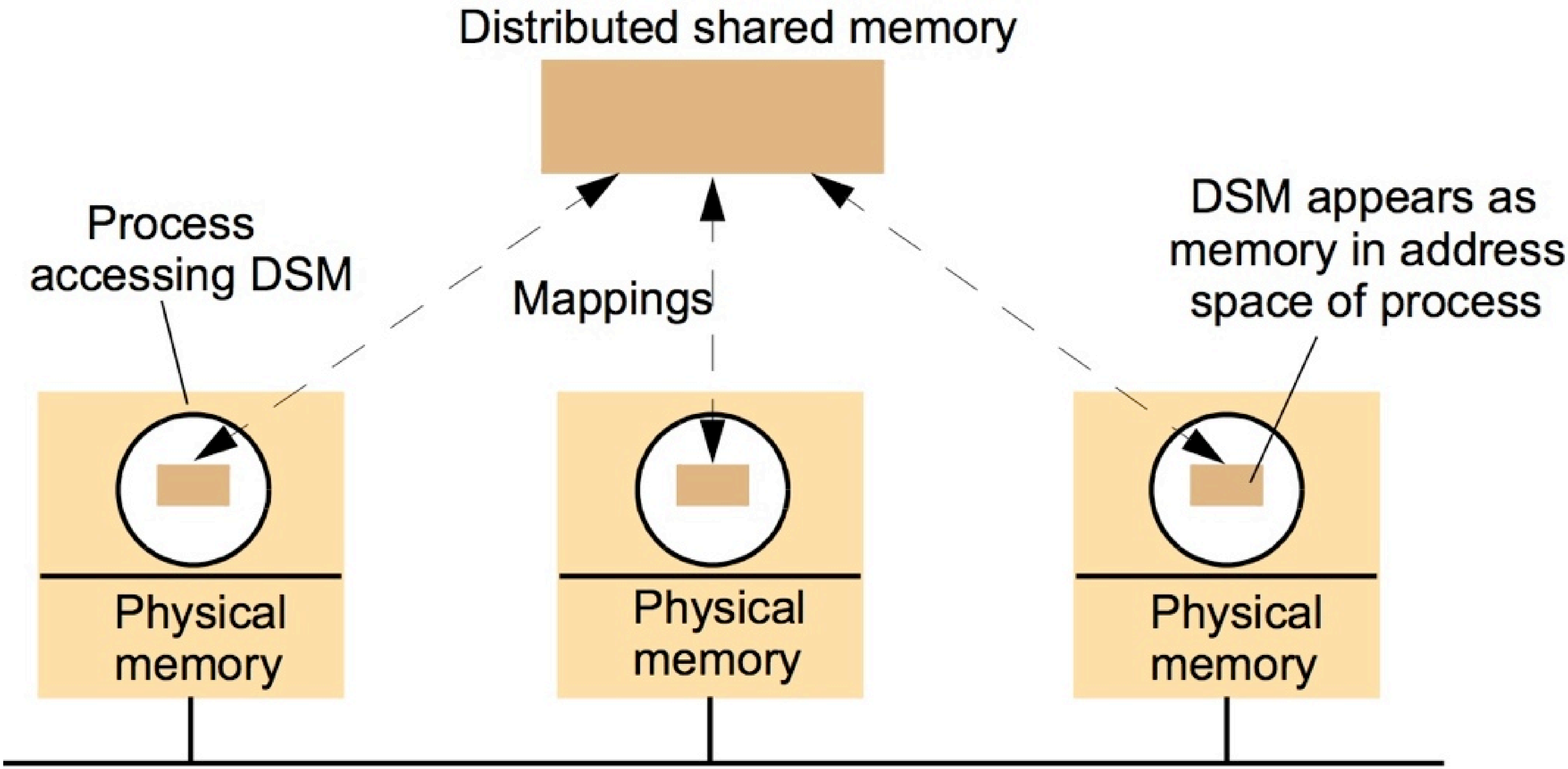
```
import javax.jms.*;  import javax.naming.*;
public class FireAlarmConsumerJMS  // similar to producer!
public String await() {
    try {
        Context ctx = new InitialContext();
        TopicConnectionFactory topicFactory =
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
         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;
   }
  } // await()
} // FireAlarmConsumerJMS – this missing in book!
```


© Pearson Education 2005

Shared memory approaches [6.5]

- Abstraction: memory locations then tuple space
- Distributed shared memory (DSM) [6.5.1]
  - Read and write with API “like” ordinary memory
  - Updates propagated by the runtime system of the DSM
  - Mostly for parallel apps or if data items can be directly accessed
  - Not as appropriate for client-server
  - Replicas of data kept & managed (problems: replication, caching)
  - Can be very useful in non-uniform access (NUMA) parallel comp’s
  - Memory space can be persistent

Figure 6.19
The distributed shared memory abstraction



© Addison-Wesley Publishers 2000

Message passing (MP) compared to DSM

- Both are lower-level than client-server or pub-sub
- Service offered
  - MP:
    - variables have to be marshalled by apps
    - Producers and consumers protected from each other (no shared memory)
  - DSM:
    - No marshalling (implications?)
    - Supports pointers
    - No app-level synchronization: DSM runtime takes care of
    - Persistent DSM supports temporal decoupling
- Efficiency
  - DSM peformance varies widely, including access patterns
  - DSM can hide the fact that something is remote (good or bad?)

Tuple space communication [6.5.2]

- A tuple is an ordered list of type values
- Tuple space is an (unordered) bag of tuple
- Can withdraw based on a specified value (or any value)
- Primitives added
  - **out**(“Subtask”, velocity, i, j, k)
  - **in**(“subtask”, ?myVelocity, ?row, 3, ?factor)
  - **rd**(“subtask”, ?myVelocity, ?row, 3, ?factor)

- Journal paper from Bakken’s dissertation cited on Page 268 of CDKB5 (“Bakken and Schlichting [1995]”).

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

© Pearson Education 2005