

Slides for Chapter 6: Indirect Communication (First part, 6.1 to 6.3)



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

Edition 5, © Addison-Wesley 2012

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 oen 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

	<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>

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

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 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

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

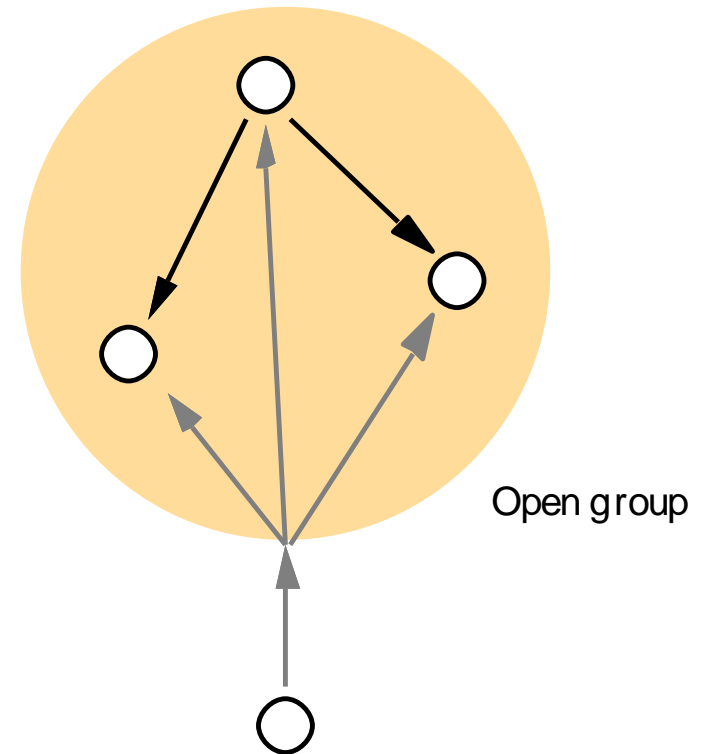
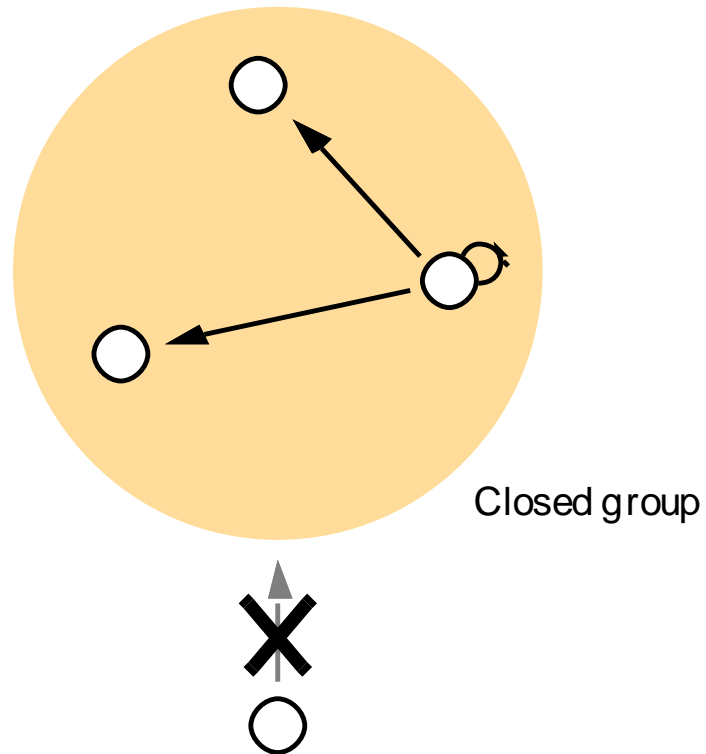
- Most research on **process groups**
 - Abstraction: resilient process
 - Messages delivered to a process endpoint, no higher
 - Messages typically unstructured byte arrays, no marshalling etc
 - Level of service \approx socket
- **Object group**: higher level approach
 - Collection of objects (same 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



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

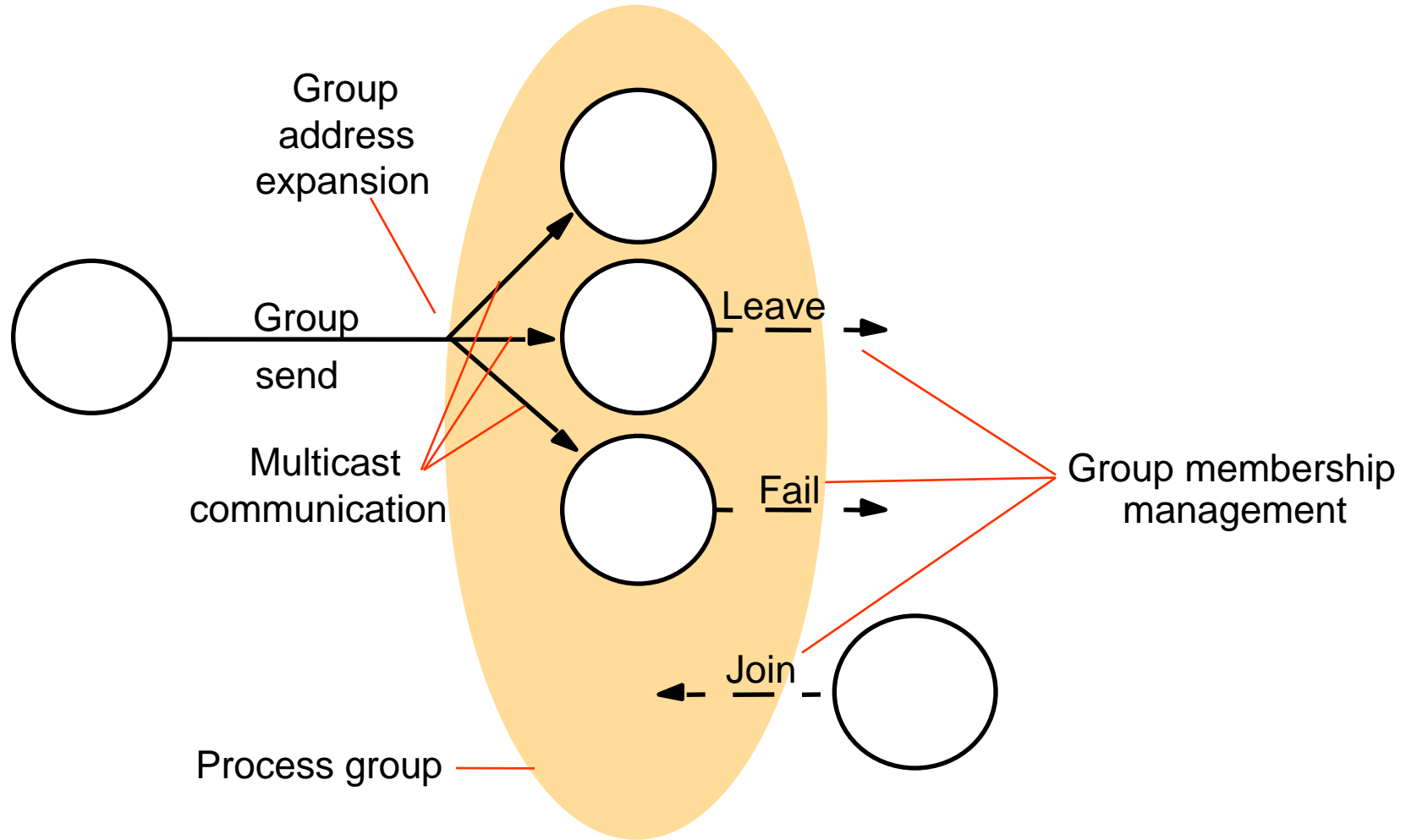
- Possible strengths of ordering
 - **FIFO ordering**: first-in-first-out from a single sender to the group
 - **Causal ordering**: preserves potential causality, happens before (Chap 14)
 - **Total ordering**: messages delivered in same order to all processes
- Perspective (not testable unless later 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

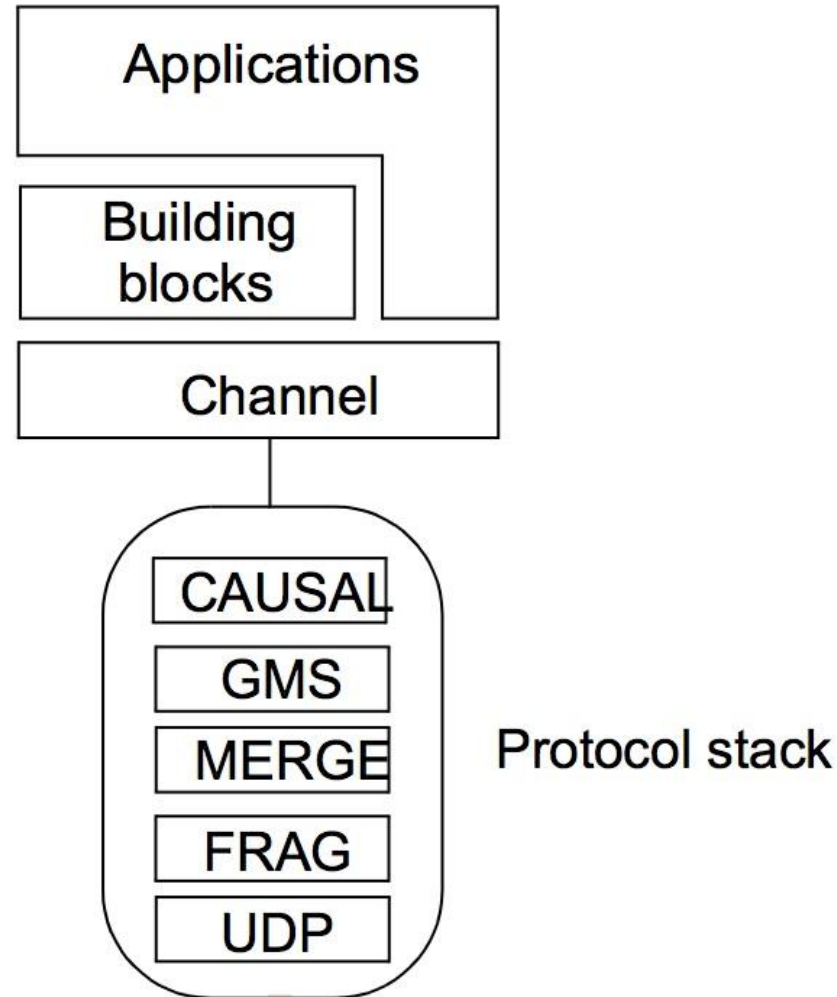


Case study: JGroups toolkit

- 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



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) {
    }
}
}
```

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;
        }
    }
}
```

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`: distributed 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]

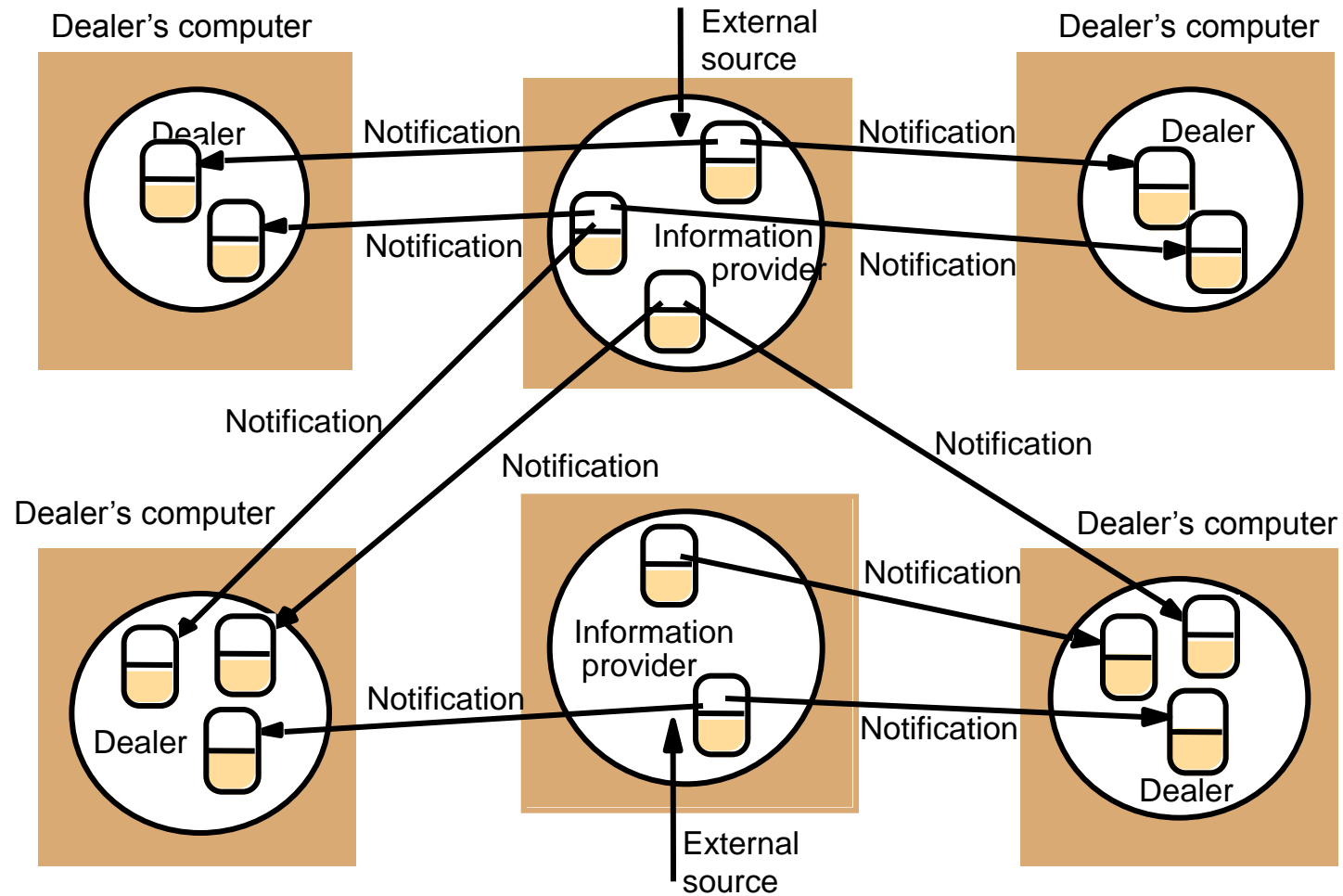
- Pub-sub AKA distributed event systems
 - Most widely used from this chapter
 - Publishers publish **structured events** to event service (ES)
 - Subscribers **express interest** in particular events
 - ES matches published events to subscriptions
- Applications (lots...)
 - Financial info systems
 - 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



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
 - All subscribers receive all events (atomicity)
 - Real-time
 - ...

Pub-sub programming model

- Publishers

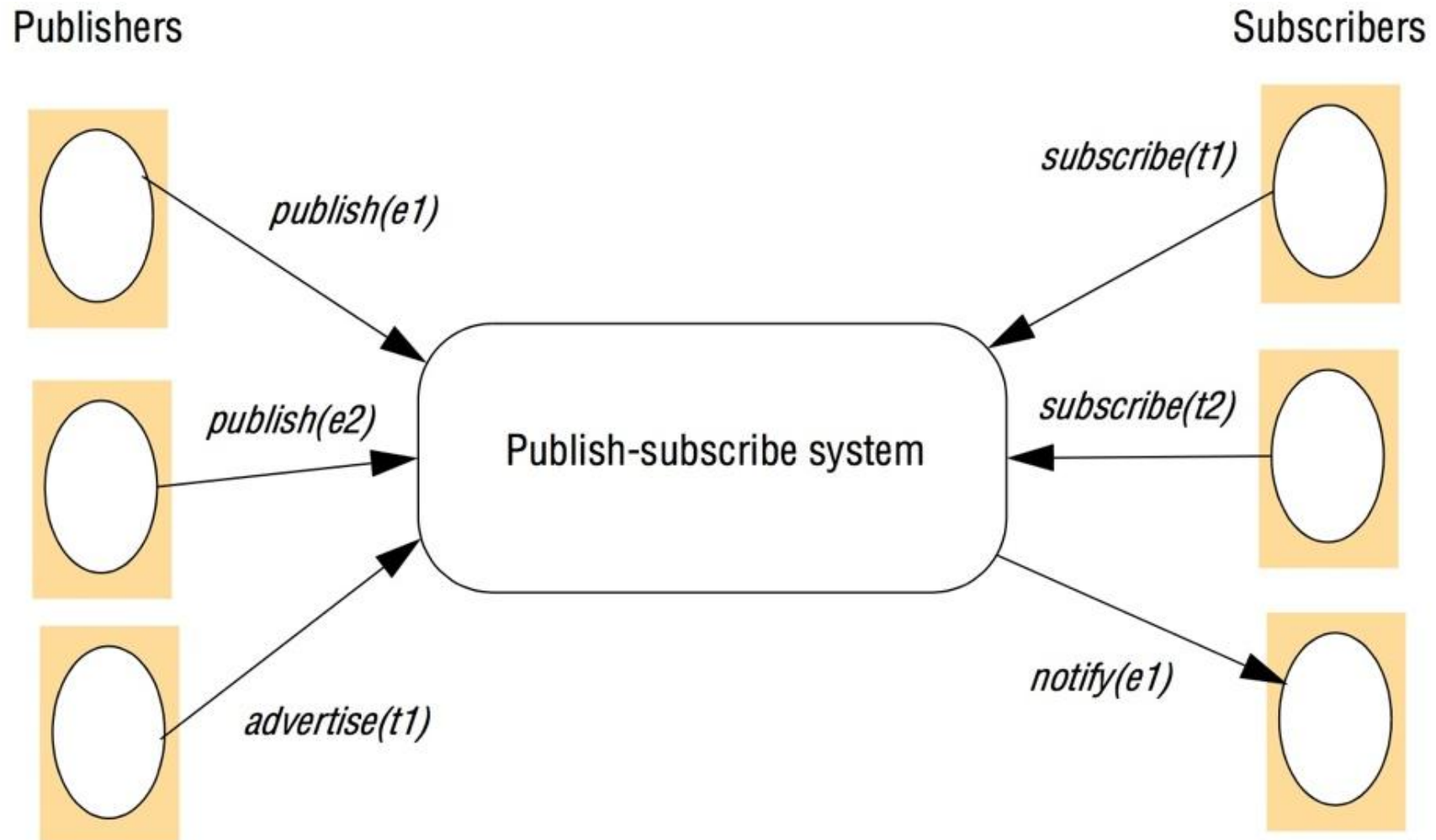
- Disseminate event e through `publish(e)`
- (Sometimes, fancier) register/advertise via a filter (pattern over all events) `f : advertise (f)`
- Expressiveness of pattern is the **subscription model** (later slide)
- Can also remove the offer to publish: `unadvertise (f)`

- 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



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.)

- 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

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

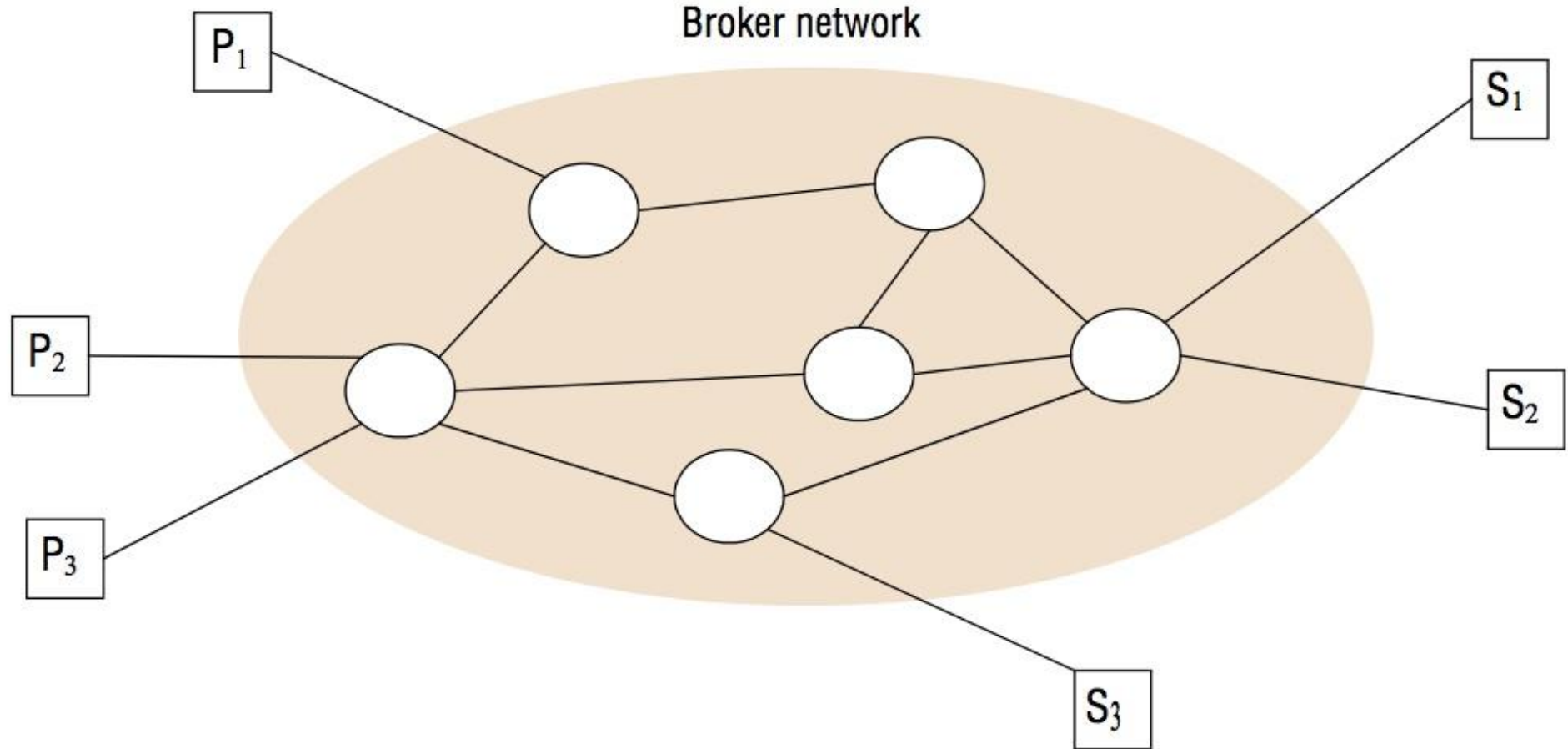
- Simple way: single centralized broker node
- **Q: Limitations?**
- Most implementations are **network of brokers**
 - E.g., GridStat
- Some implementations are peer-to-peer (P2P)
 - All publisher and subscriber nodes act as the pub-sub broker
 - E.g., RTI DDS
- **Q: Plusses and minuese of network of brokers vs. P2P?**

Figure 6.9

A network of brokers

Publishers

Subscribers

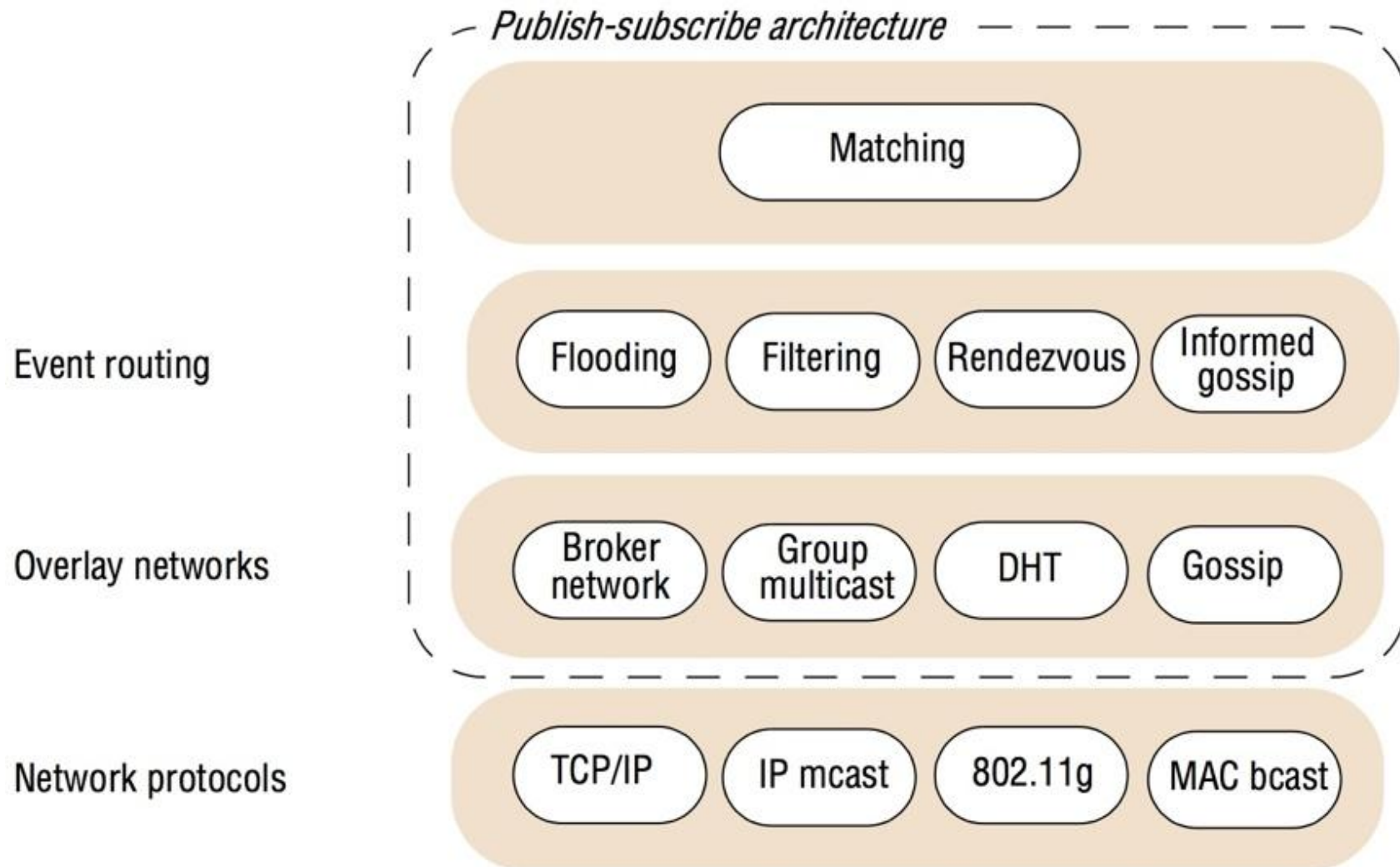


Overall systems architecture

- 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
- 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 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

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**: $SN(s) \cap EN(e)$ 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

<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