Indirect Communication

Agenda

Group communication

Publish-subscribe systems

Message queues

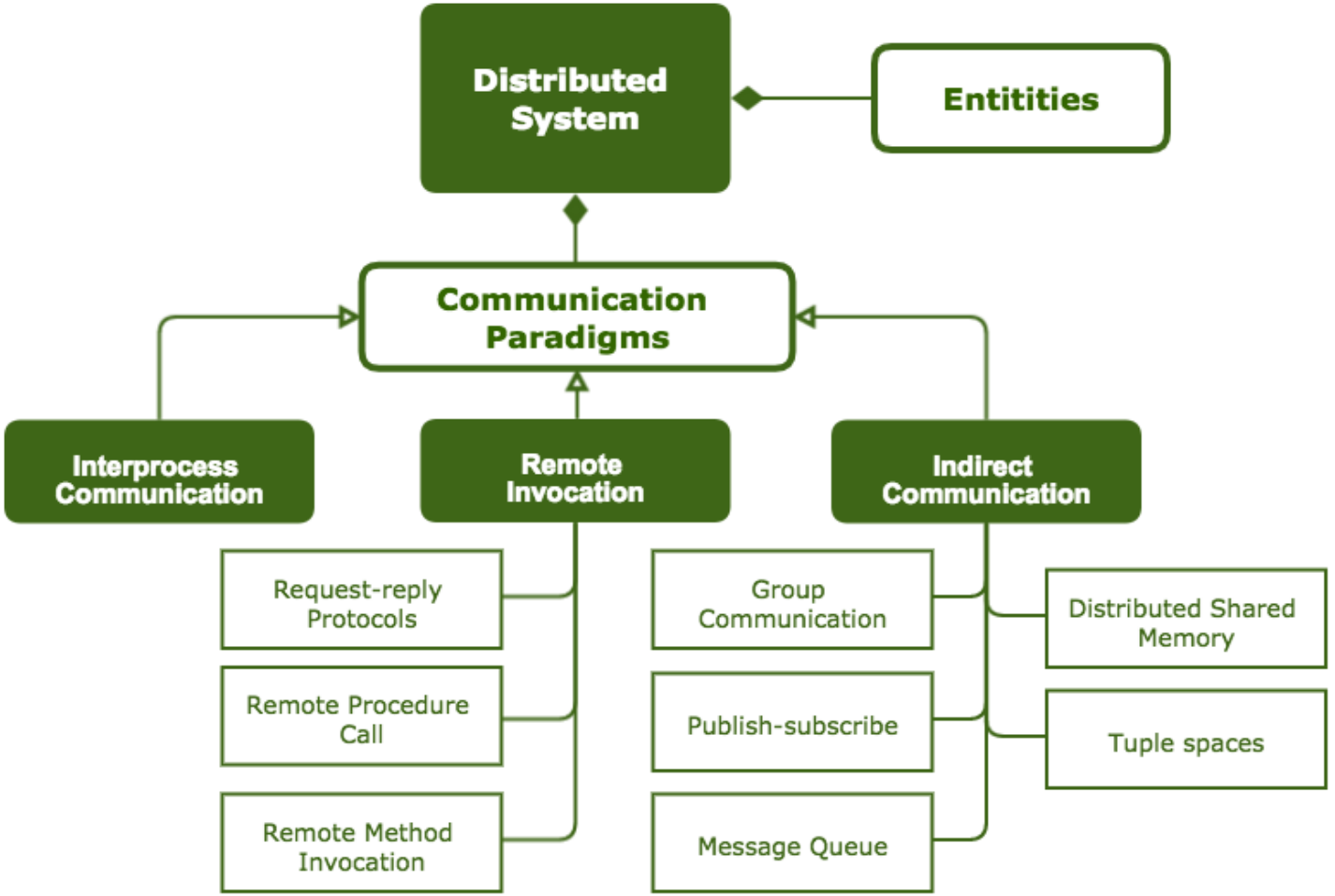
Shared memory approaches

Interprocess communication

Interprocess communication is at the heart of all distributed systems

- We study the ways that processes on different machines can exchange information
- Programming by message passing is harder than using primitives based on shared memory, as available for non-distributed platforms
- Asynchronus communication: when it cannot be assumed that the receiving side is active at the time a request is issued, alternative time-uncoupled communication services are needed

Communication paradigms in distributed systems



https://dzone.com/articles/message-brokers-in-indirect-communication-paradigm

Indirect communication

The essence of indirect communication is to communicate through an intermediary and have no direct coupling between the sender and the receiver(s)

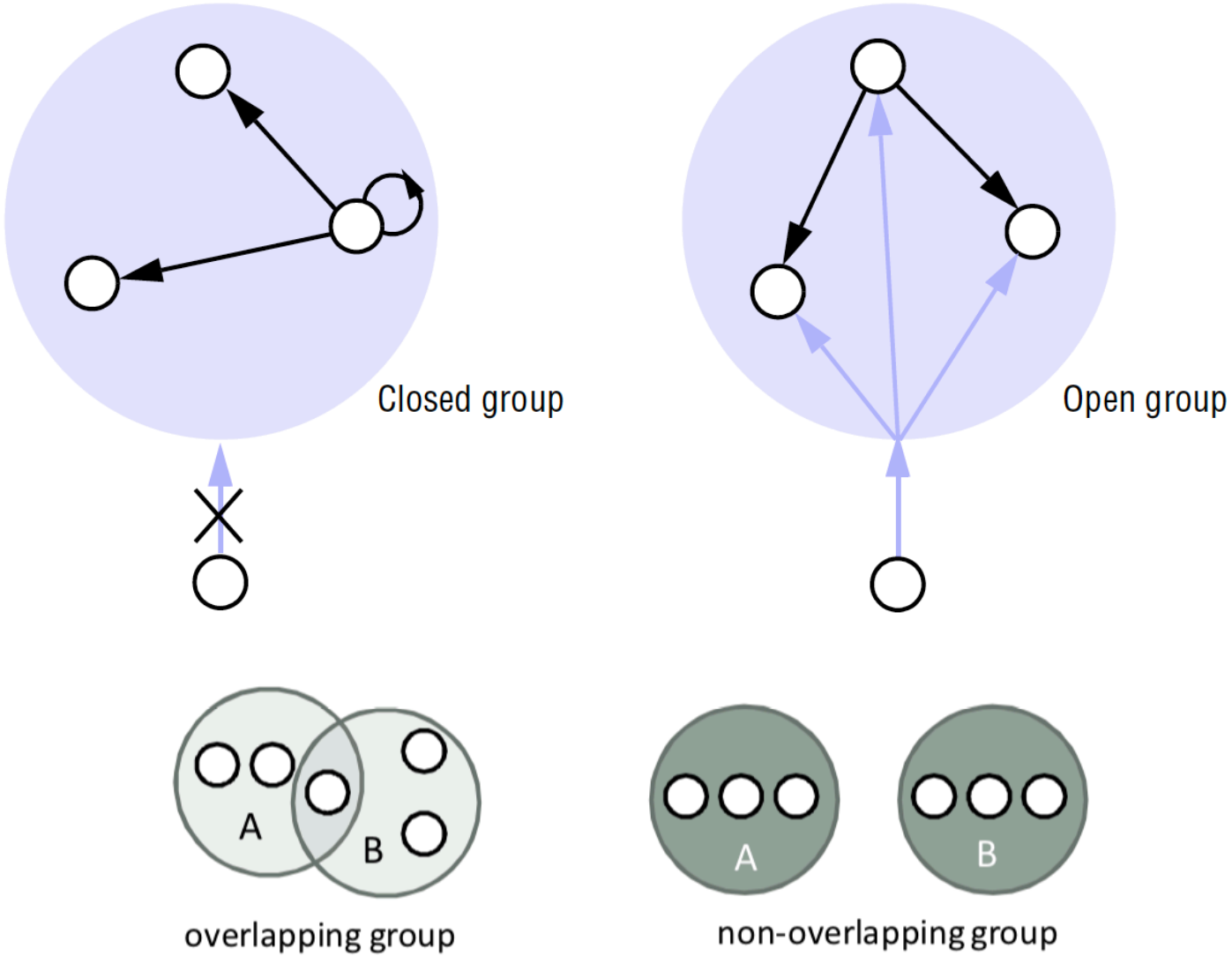
The concepts of *space and time uncoupling* are useful for:

- group communication, in which communication is via a group abstraction with the sender unaware of the identity of the recipients;
- publish-subscribe systems, a family of approaches that all share the common characteristic of disseminating events to multiple recipients through an intermediary;
- message queue systems, wherein messages are directed to the abstraction of a queue with receivers extracting messages from such queues;
- shared memory–based approaches, including distributed shared memory and tuple space approaches, which present to programmers an abstraction of a global shared memory

Space and time coupling in distributed systems

	Time coupled	Time uncoupled
Space coupling	Properties: Communication directed to given receivers that must exist at that moment in time Examples: message passing, remote invocation	Properties: Communication directed to given receivers; senders and receivers can have non overlapping Exampe lifetimes
Space uncoupling	Properties: Sender does not need to know the identity of the receivers; senders and receivers must exist at that moment in time Example: IP multicast	Properties: Sender does not need to know the identity of the receivers; senders and receivers can have non overlapping lifetimes Examples: see next slides

Open, closed and overlapping groups



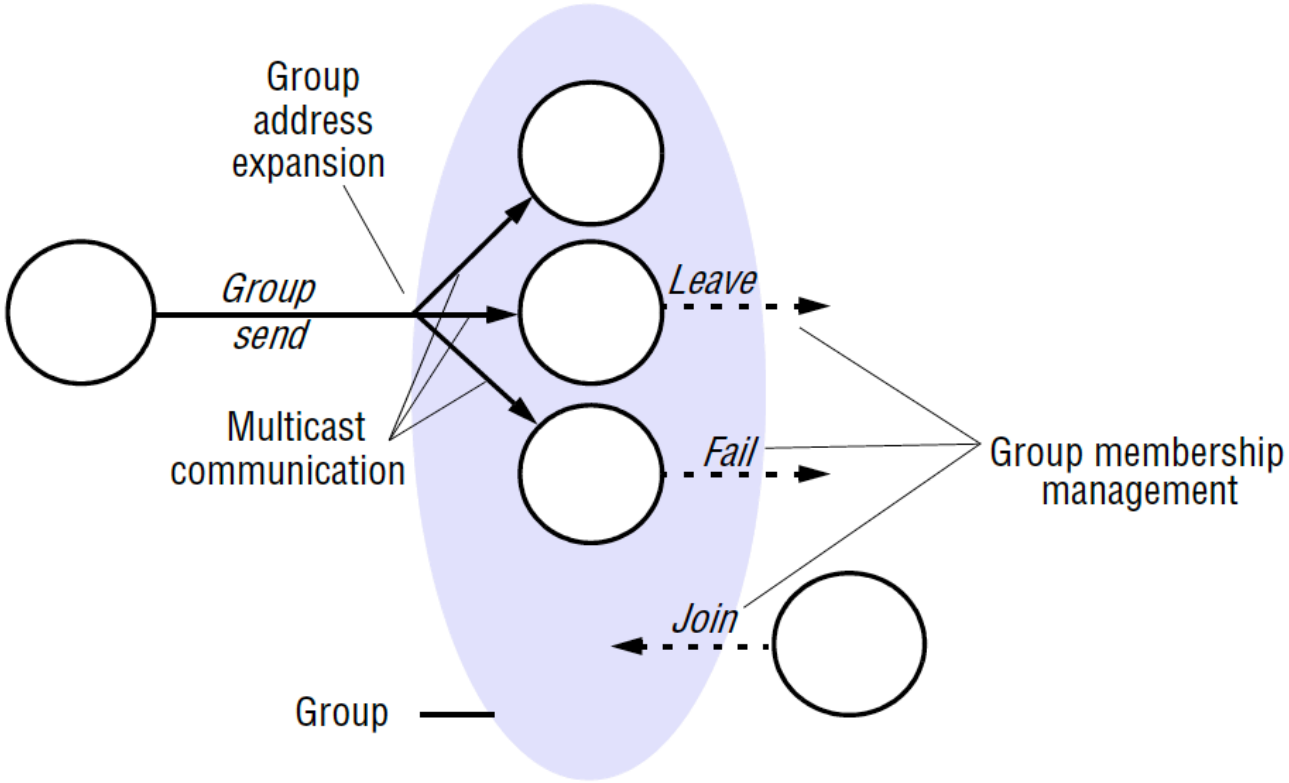
Ordering multicast messages

Ordering is not guaranteed by underlying interprocess communication primitives.

Group communication services offer ordered multicast, with the option of one or more of the following properties (with hybrid solutions also possible):

- FIFO ordering : (also referred to as source 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 at all processes in the group.
- Causal ordering : Causal ordering takes into account causal relationships between messages, in that if a message happens before another message in the distributed system this so-called causal relationship will be preserved in the delivery of the associated messages at all processes (we have to define precisely 'happens before').
- Total ordering : In total ordering, if a message is delivered before another message at one process, then the same order will be preserved at all processes

The role of group membership management



Group membership service

A group membership service has four main tasks:

- Providing an interface for group membership changes : The membership service provides operations to create and destroy process groups and to add or withdraw a process to or from a group. In most systems, a single process may belong to several groups at the same time (overlapping groups, as defined above). This is true of IP multicast, for example.
- 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. The detector marks processes as Suspected or Unsuspected . The service uses the failure detector to reach a decision about the group's membership: it excludes a process from membership if it is suspected to have failed or to have become unreachable.
- Notifying members of group membership changes : The service notifies the group's members when a process is added, or when a process is excluded (through failure or when the process is deliberately withdrawn from the group).
- Performing group address expansion : When a process multicasts a message, it supplies the group identifier rather than a list of processes in the group. The membership management service expands the identifier into the current group

ISIS (by Ken Birman @ Cornell University)

- Birman is known for developing the Isis Toolkit, which introduced the virtual synchrony execution model for multicast communication.
- Birman founded Isis Distributed Systems to commercialize this software, which was used by stock exchanges, for air traffic control, and in factory automation.
- The Isis software operated the New York and Swiss Stock Exchanges for more than a decade, and continues to be actively used in the French air traffic control system and the US Navy AEGIS warship
- The technology permits distributed systems to automatically adapt themselves when failures or other disruptions occur, to securely share keys and security policy data, and to replicate critical services so that availability can be maintained even while some system components are down. Birman released a version of the Isis technology, Vsync, as an open-source library

The Virtual Synchrony model

- Virtual synchrony is an interprocess message passing (sometimes called ordered, reliable multicast) technology.
- Virtual synchrony systems allow programs running in a network to organize themselves into process groups, and to send messages to groups (as opposed to sending them to specific processes).
- Each message is delivered to all the group members, in the identical order, and this is true even when two messages are transmitted simultaneously by different senders.
- Application design and implementation is greatly simplified by this property: every group member sees the same events (group membership changes and incoming messages) in the same order.

Implementing the Virtual Synchrony model

- A virtually synchronous service is typically implemented using a style of programming called state machine replication, in which a service is first implemented using a single program that receives inputs from clients through some form of remote message passing infrastructure, then enters a new state and responds in a deterministic manner.
- The initial implementation is then transformed so that multiple instances of the program can be launched on different machines, using a virtually synchronous message passing system to replicate the incoming messages over the members.
- The replicas will see the same events in the same order, and are in the same states, hence they will make the same state transitions and remain in a consistent state.

Jgroups

- JGroups is a library for reliable one-to-one or one-to-many communication written in the Java language.
- It can be used to create groups of processes whose members send messages to each other.
- JGroups enables developers to create reliable multipoint (multicast) applications where reliability is a deployment issue.
- JGroups also relieves the application developer from implementing this logic themselves.
- This saves significant development time and allows for the application to be deployed in different environments without having to change code.

Jgroups (http://www.jgroups.org)

Jgroups can be used to create clusters whose nodes can send messages to each other. Its main features include:

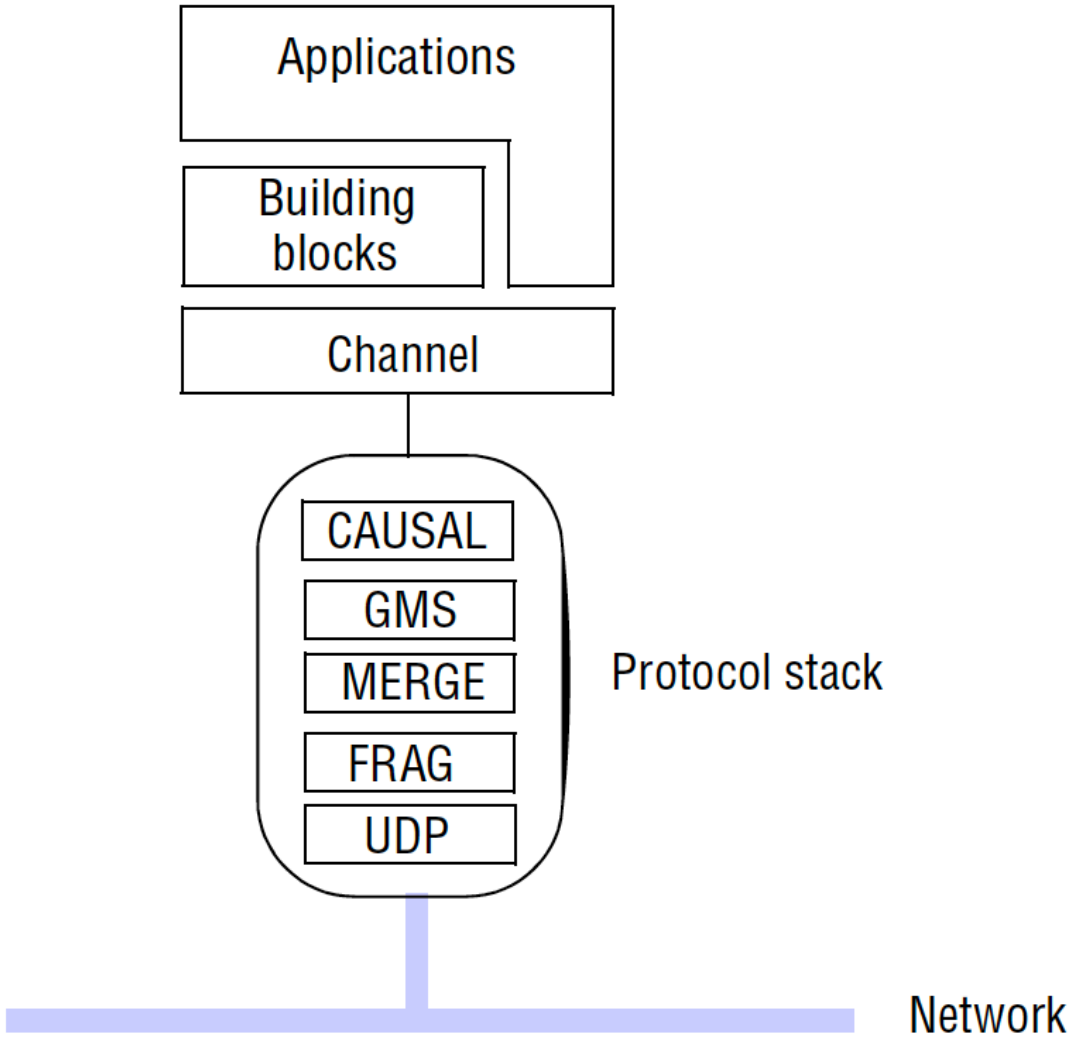
- Cluster creation and deletion.
- Cluster nodes can be spread across LANs or WANs
- Joining and leaving of clusters
- Membership detection and notification about joined/left/crashed cluster nodes
- Detection and removal of crashed nodes
- Sending and receiving of node-to-cluster messages (point-to-multipoint)
- Sending and receiving of node-to-node messages (point-to-point)

JGroups

The main components of the JGroups implementation:

- Channels represent the most primitive interface for application developers, offering the core functions of joining, leaving, sending and receiving.
- Building blocks offer higher-level abstractions, building on the underlying service offered by channels.
- A protocol stack provides the underlying communication protocol, constructed as a stack of composable protocol layers (see next slide)

The architecture of Jgroups (Jgroups.org)



Jgroups protocol stack layers

The protocol stack consists of five layers:

- The UDP layer is the most common transport layer in JGroups. this is not entirely equivalent to the UDP protocol; rather, the layer utilizes IP multicast for sending to all members in a group and UDP datagrams specifically for point-to-point communication. This layer assumes that IP multicast is available.
- FRAG implements message packetization and is configurable in terms of the maximum message size (8,192 bytes by default).
- MERGE is a protocol that deals with unexpected network partitioning and the subsequent merging of subgroups after the partition. A series of alternative merge layers are actually available, ranging from the simple to ones that deal with, for example, state transfer.
- GMS implements a group membership protocol to maintain consistent views of membership across the group
- CAUSAL implements causal ordering

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) {
    }
}
```

Java class *FireAlarmConsumerJG*

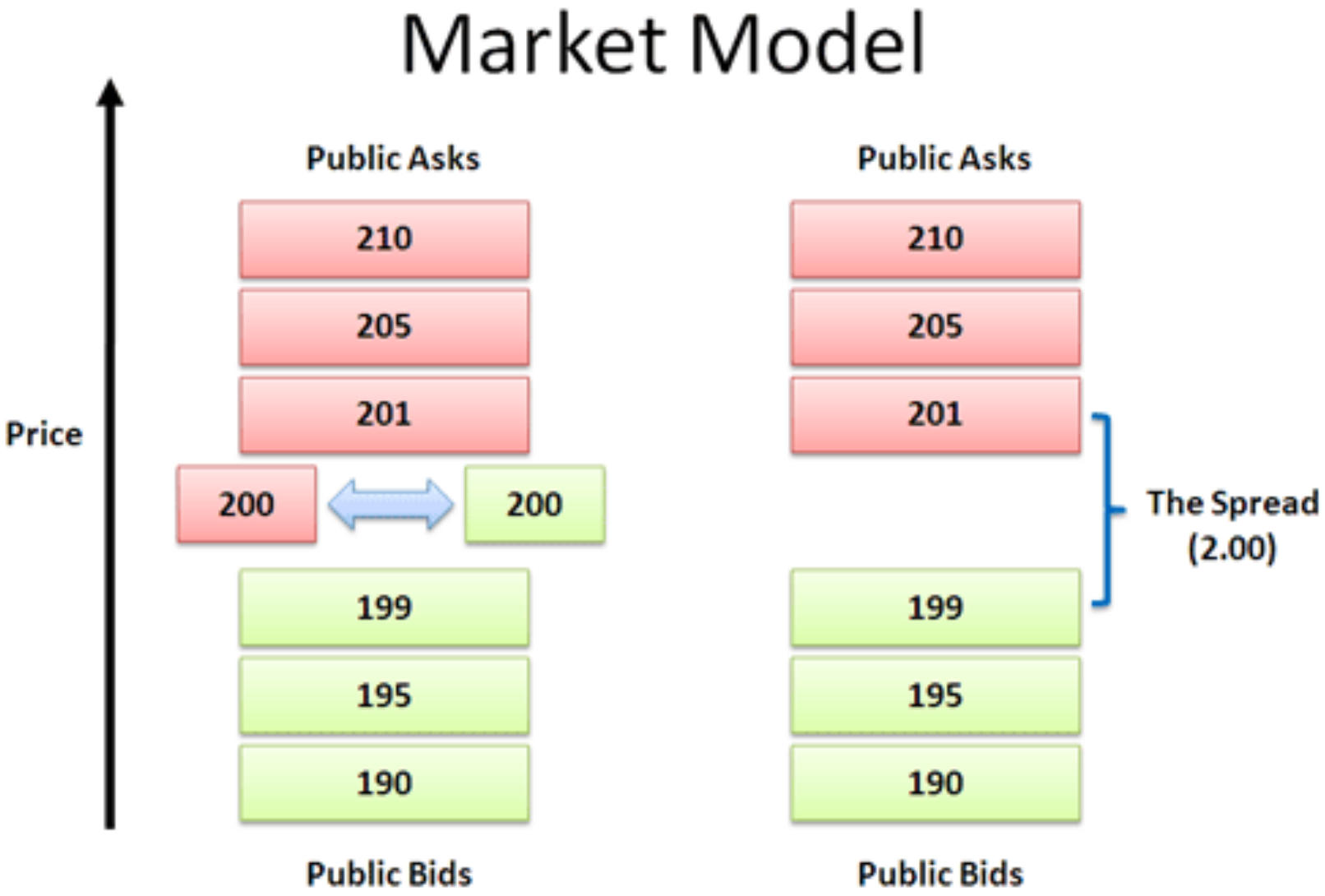
```
import org.jgroups.JChannel;

   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;
       }
   }
}
```

Publish subscribe systems

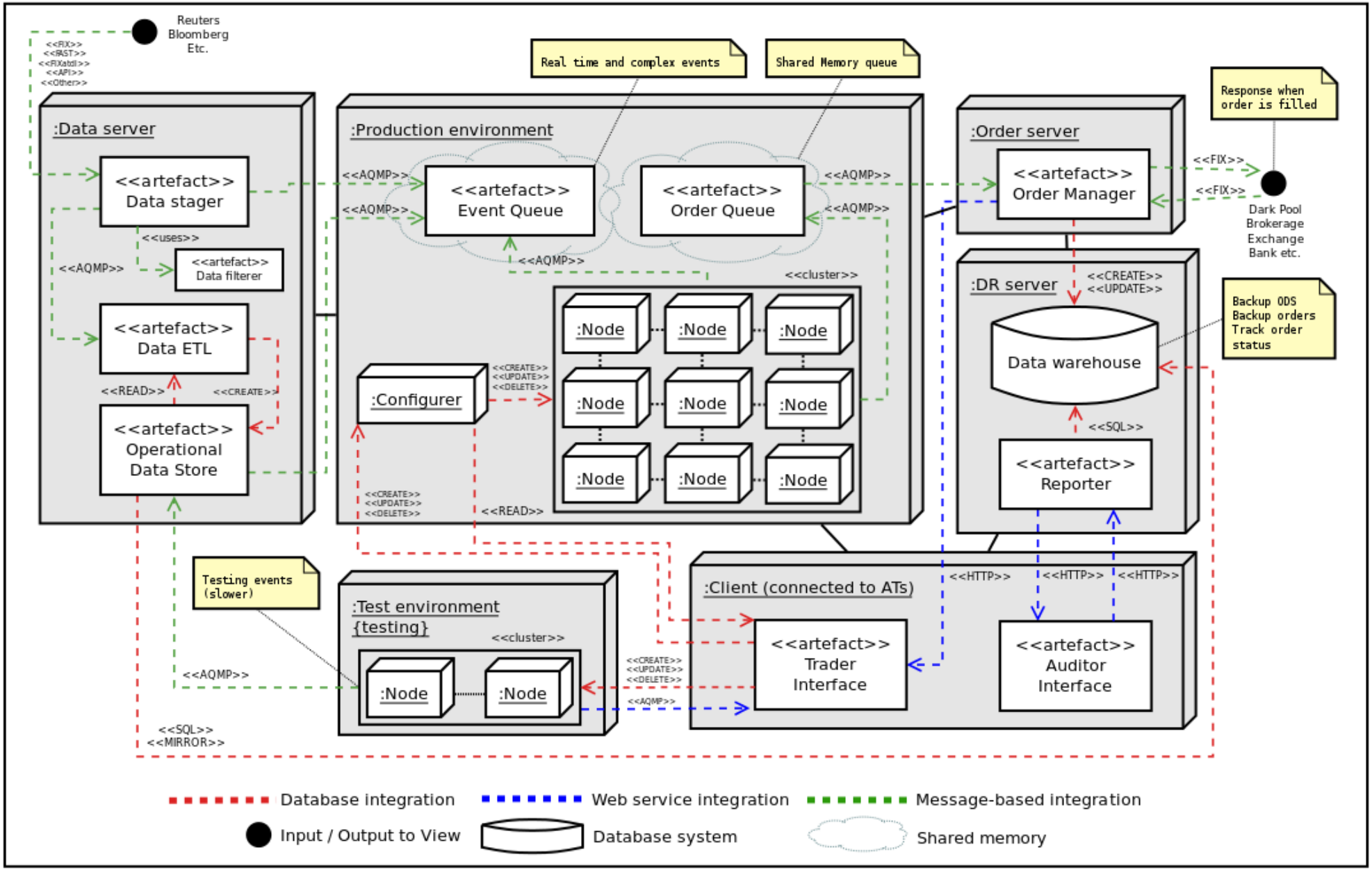
Publish-subscribe systems have three main features:

- Heterogeneity: components designed independently may work together
- Asynchronicity: publishers and subscribers are decoupled
- A variety of different delivery guarantees can be provided for notifications - the one that is chosen should depend on the application requirements

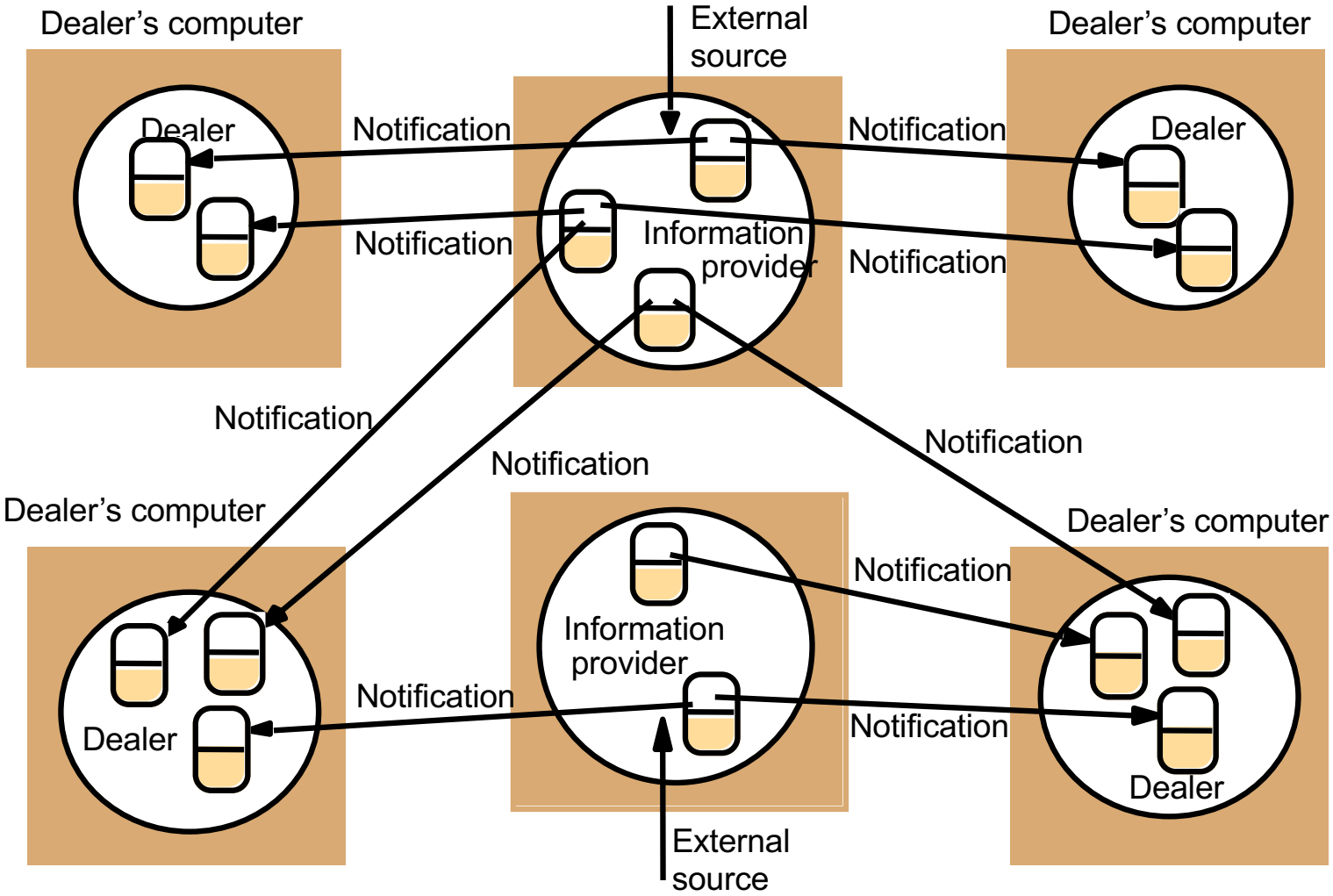
Stock exchange: an example of distributed system



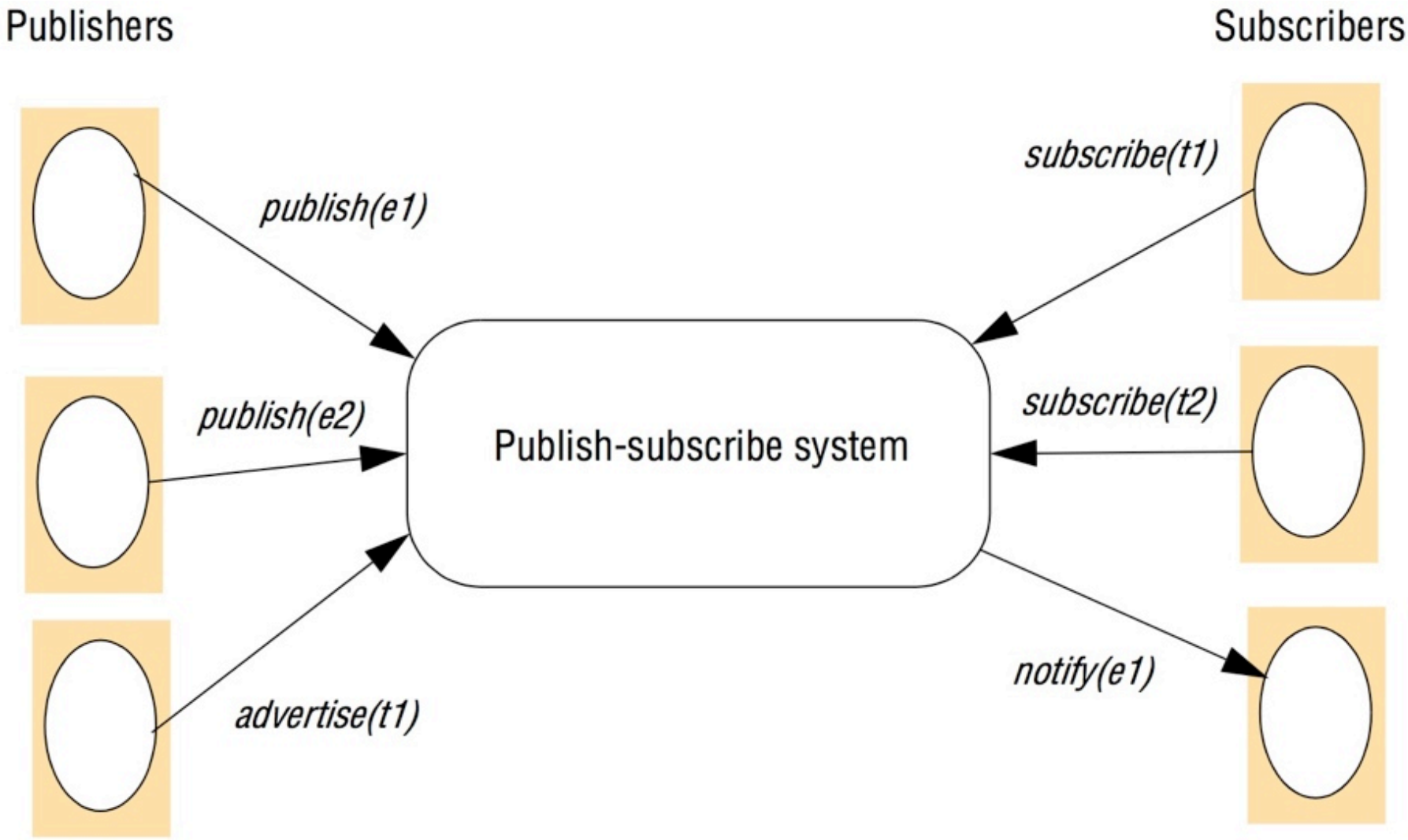
Algorithmic Trading system (ATs) High Level Deployment View



Dealing room system



The publish-subscribe paradigm



Variants of publish subscribe schemes

Channel-based:

In this approach, publishers publish events to named channels and subscribers then subscribe to one of these named channels to receive all events sent to that channel.

Example: CORBA event service

Variants of publish subscribe schemes

Topic-based (also referred to as subject-based):

In this approach, each notification is expressed in terms some *fields*, with one field denoting the topic.

Subscriptions are then defined in terms of the topic of interest.

This is equivalent to channel-based approaches, with the difference that topics are implicitly defined in the case of channels but explicitly declared in topic-based approaches

Variants of publish subscribe schemes

Content-based: Content-based approaches are a generalization of topic-based approaches allowing the expression of subscriptions over a range of fields in an event notification.

A content-based filter is a query defined in terms of compositions of constraints over the values of event attributes.

For example, a subscriber could express interest in events that relate to the topic of publish-subscribe systems, where the system is the ‘CORBA Event Service’ and where the author is ‘Smith’ or ‘White’.

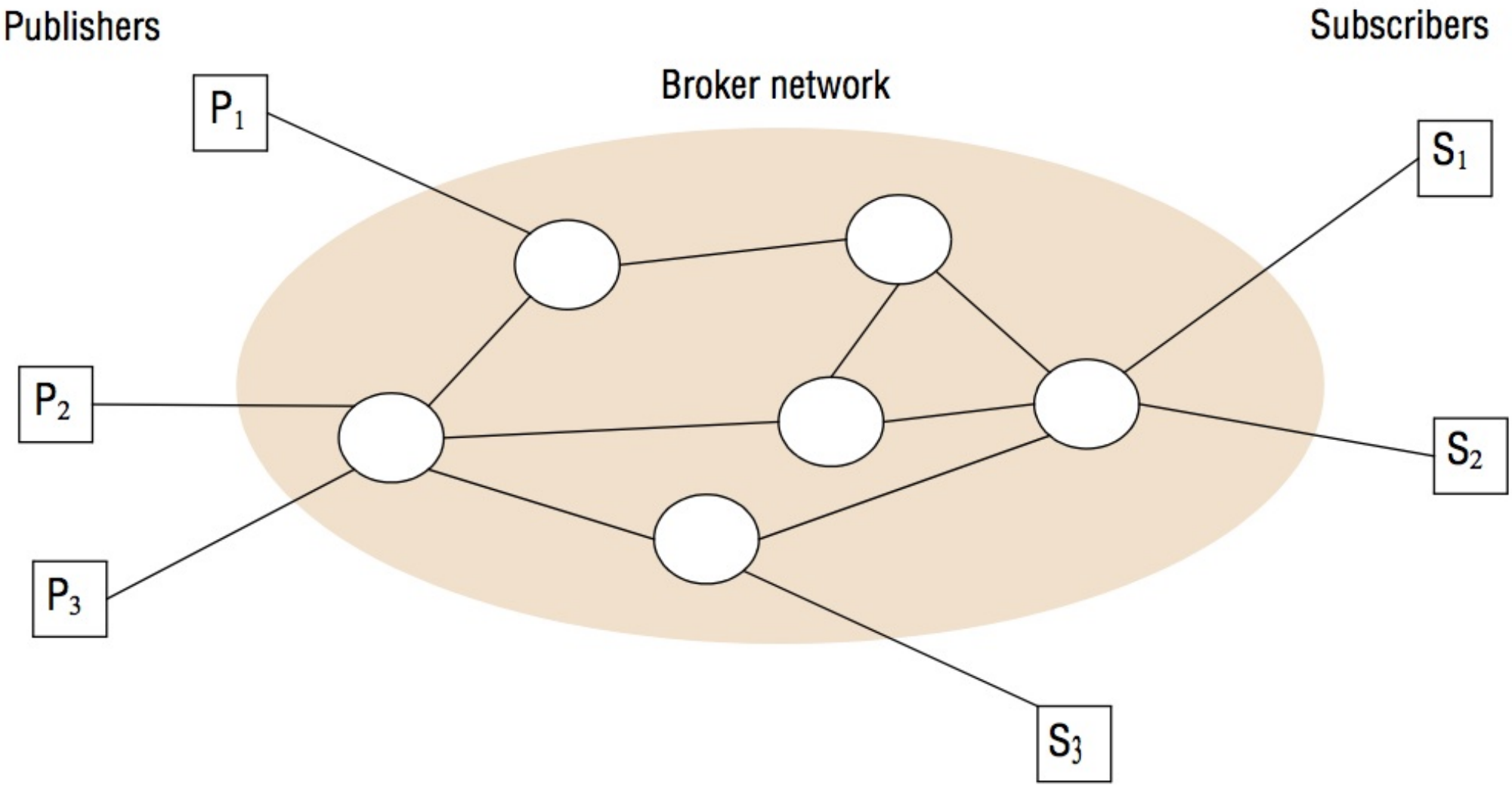
The sophistication of the associated query languages varies from system to system, but in general this approach is significantly more expressive than channel- or topic-based approaches

Variants of publish subscribe schemes

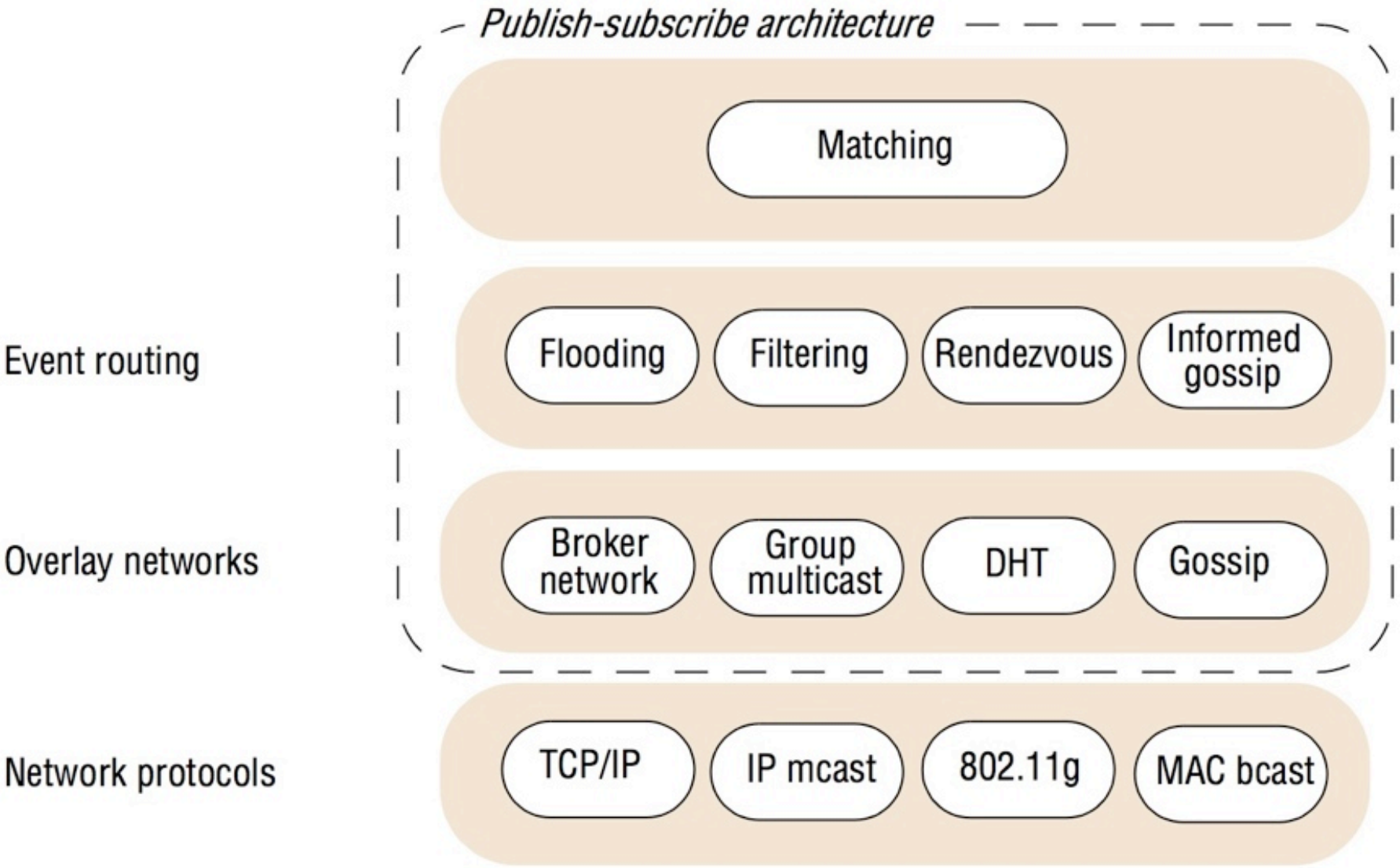
Type-based: These approaches are linked with object-based approaches where objects have a specified type.

- In type-based approaches, subscriptions are defined in terms of types of events and matching is defined in terms of types or subtypes of the given filter.
- This approach can express a range of filters, from coarsegrained filtering based on overall type names to more fine-grained queries defining attributes and methods of a given object.
- Such fine-grained filters are similar to content-based approaches.
- The advantages of type-based approaches are that they can be integrated elegantly into programming languages and they can check the type correctness of subscriptions, eliminating some kinds of subscription errors

A network of brokers



The architecture of publish-subscribe systems



Implementation approached of publish-subscribe

- Flooding: sending an event notification to all nodes
- Filtering: brokers forward notifications only to valid subscribers
- Advertisements
- Rendezvous

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
```

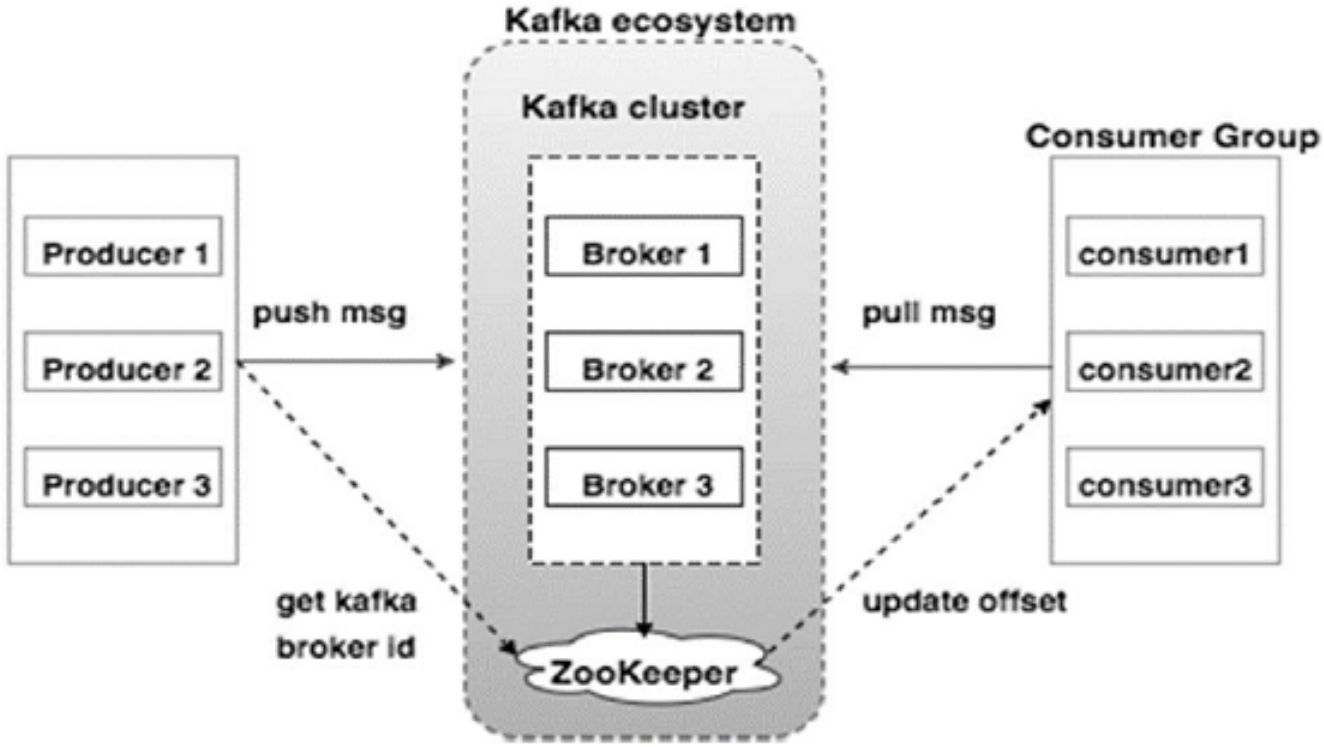
Rendezvous-based routing

```
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;
```

Some publish-subscribe systems

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

Apache Kafka



- Platform for handling real-time data feeds, written in Java and Scala
- Multiple brokers to maintain load balance
- ZooKeeper supports high availability through redundant services
- Brokers are stateless, so they use ZooKeeper for maintaining their cluster state, ZK is also used to notify producer and consumer about new or crashed broker

Message queues

- the programming model offered by message queues is simple
- It offers an approach to communication in distributed systems through queues.
- In particular, producer processes can send messages to a specific queue and other (consumer) processes can then receive messages from this queue.

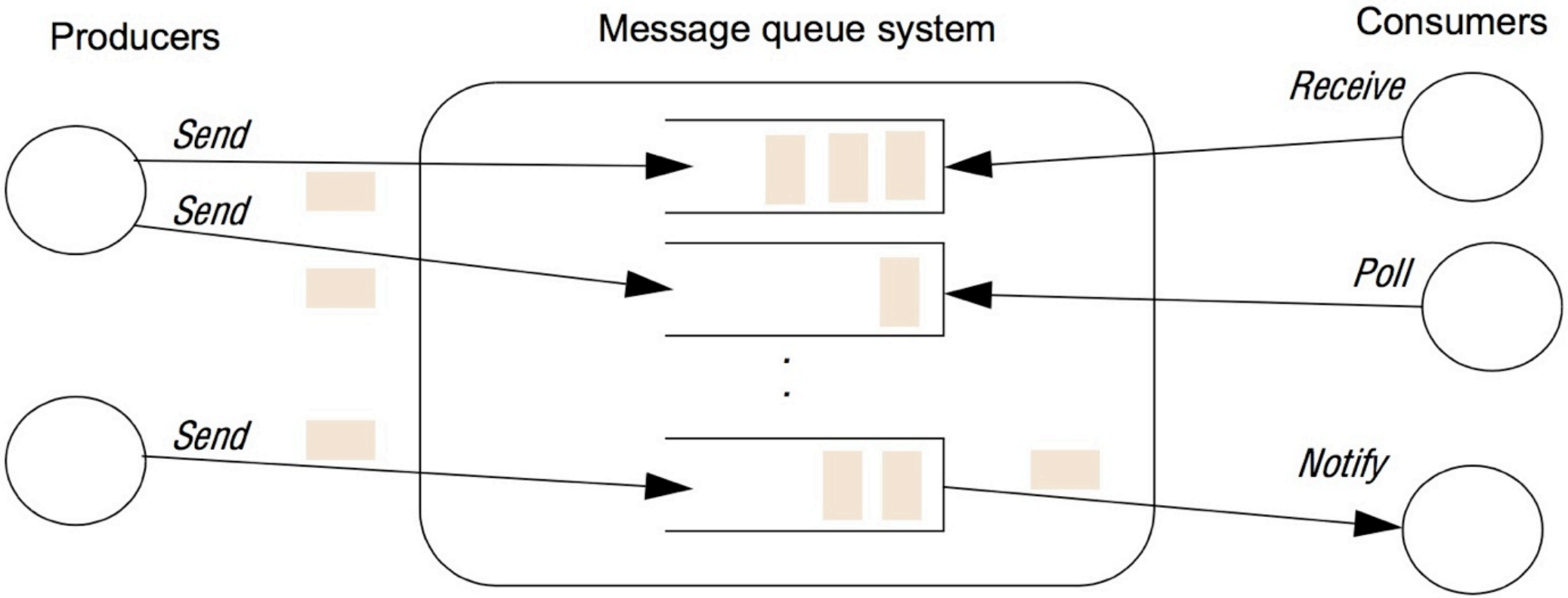
Three styles of receive are generally supported:

I. a blocking *receive*, which will block until an appropriate message is available;

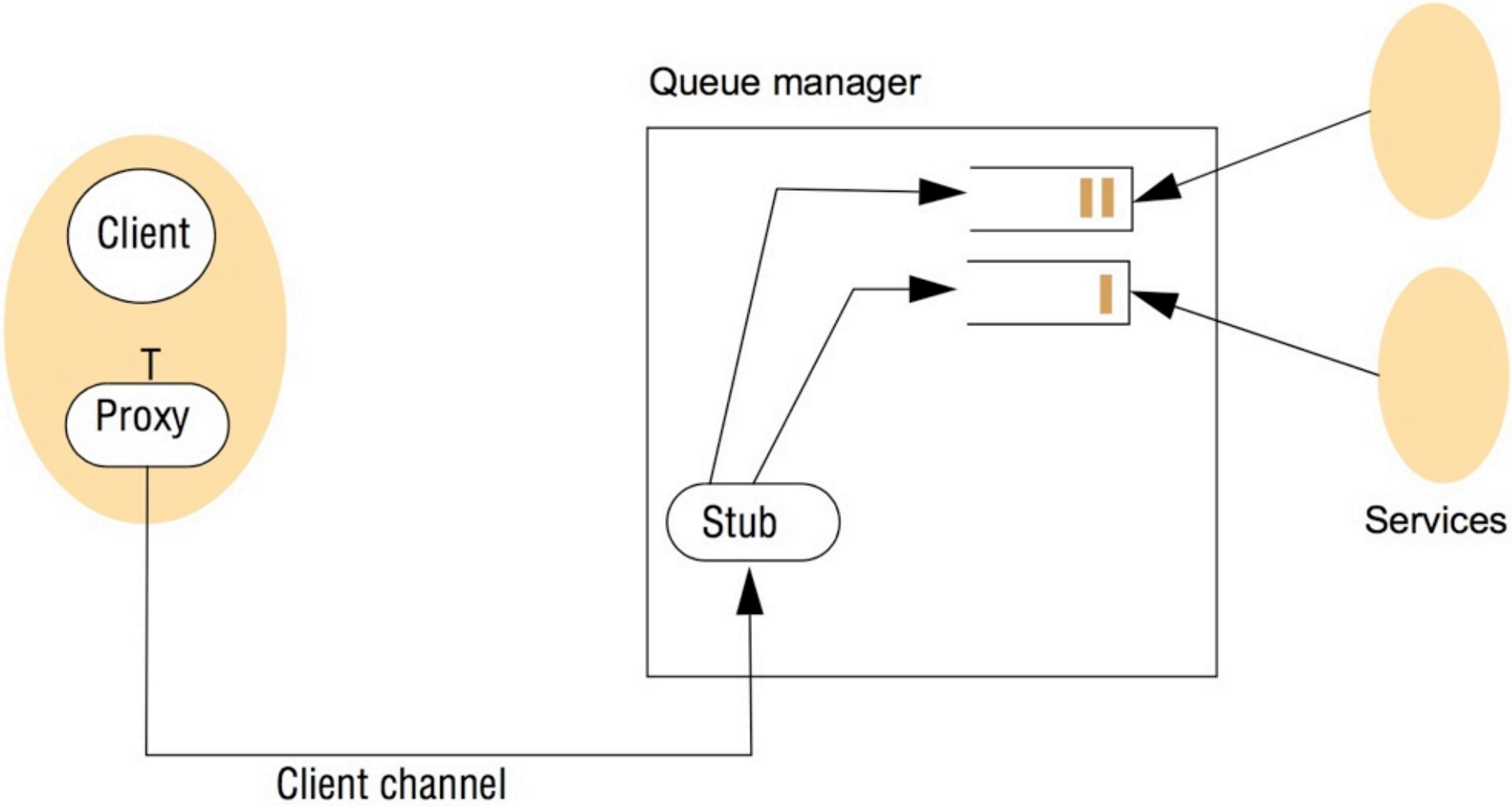
II. a non-blocking *receive* (a polling operation), which will check the status of the queue and return a message if available, or a not available indication otherwise;

III. a *notify* operation, which will issue an event notification when a message is available in the associated queue.

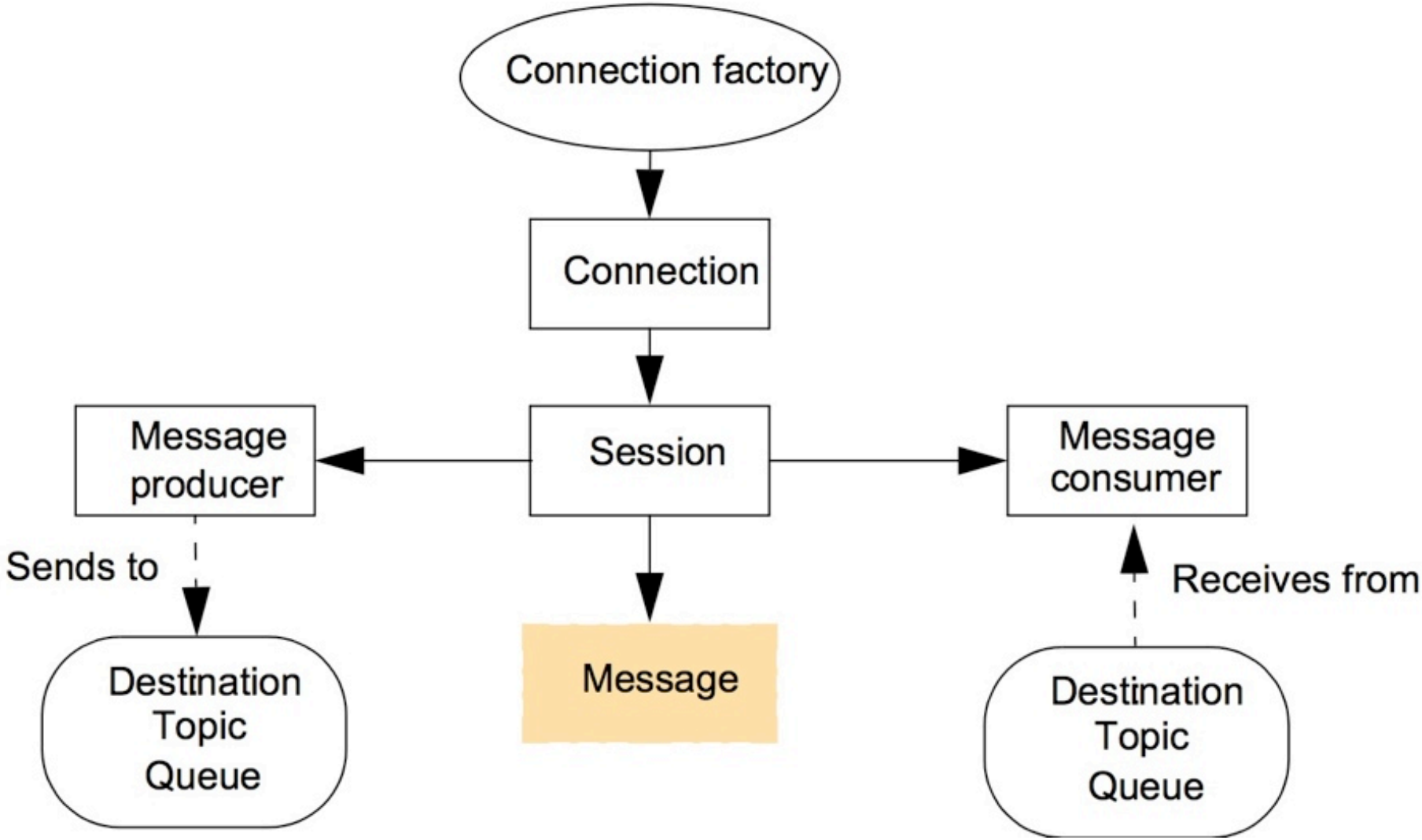
The message queue paradigm



A simple networked topology in WebSphere MQ



The programming model offered by JMS



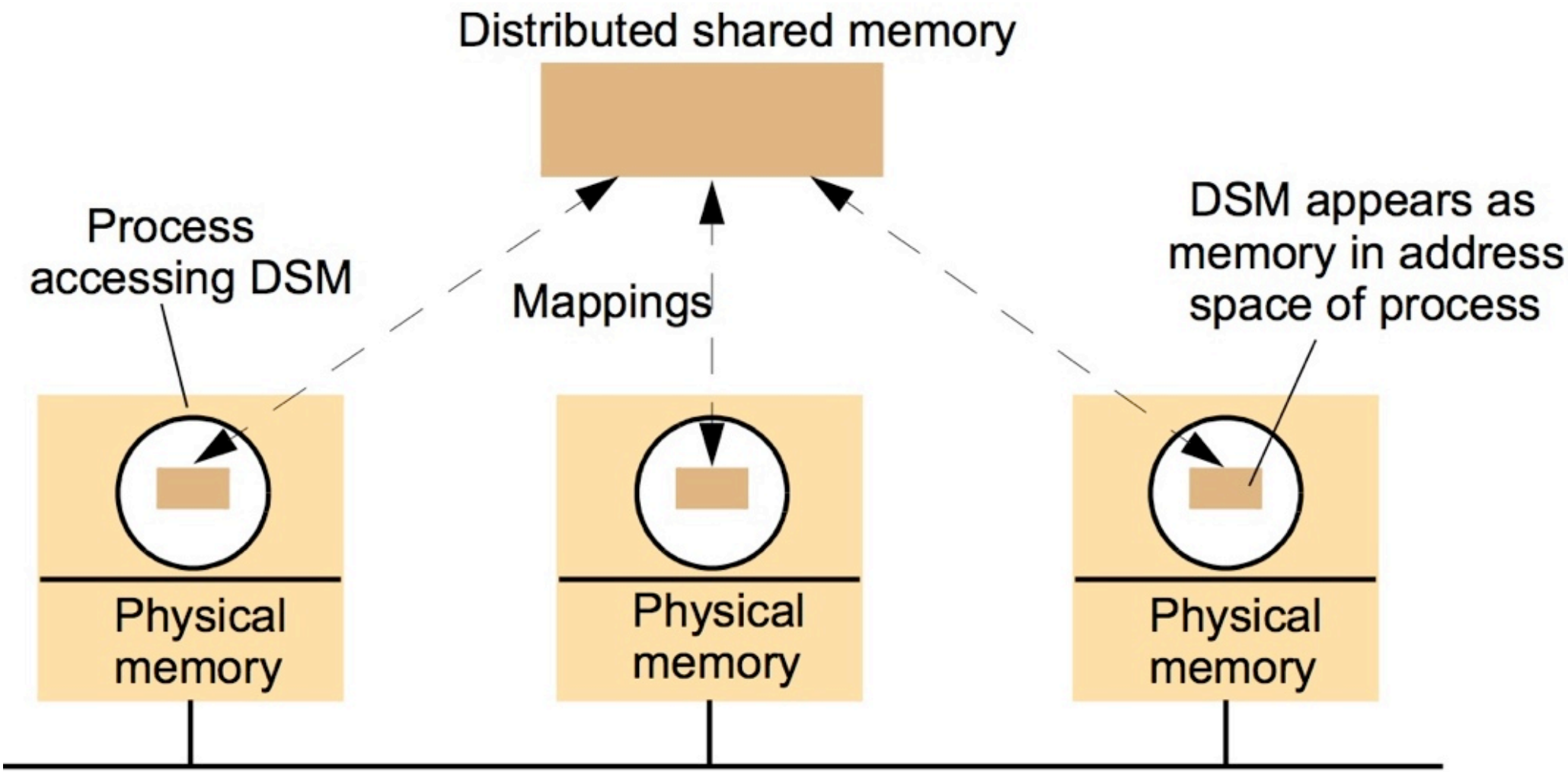
Java class *FireAlarmJMS*

```
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(message);    13
            } catch (Exception e) {     14
            } 15
}
```

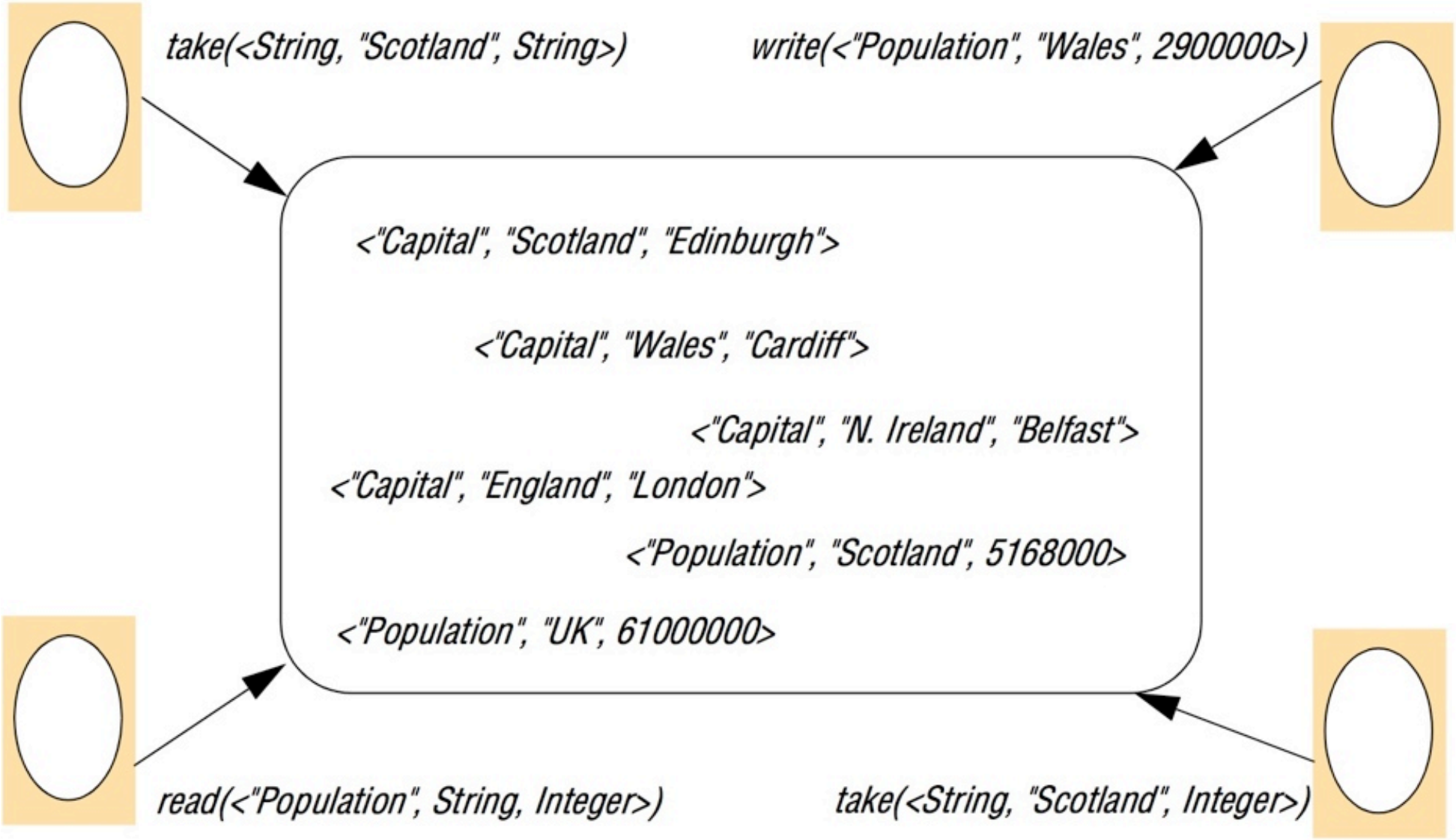
Java class *FireAlarmConsumerJMS*

```
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
}
```

The distributed shared memory abstraction



The tuple space abstraction



Replication and the tuple space operations [Xu and Liskov 1989]

write

1. The requesting site multicasts the *write* request to all members of the view;
2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

read

1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others); 4. Step 1 is repeated until at least one response is received.

continued on next slide

(continued)
Replication and the tuple space operations [Xu and Liskov 1989]

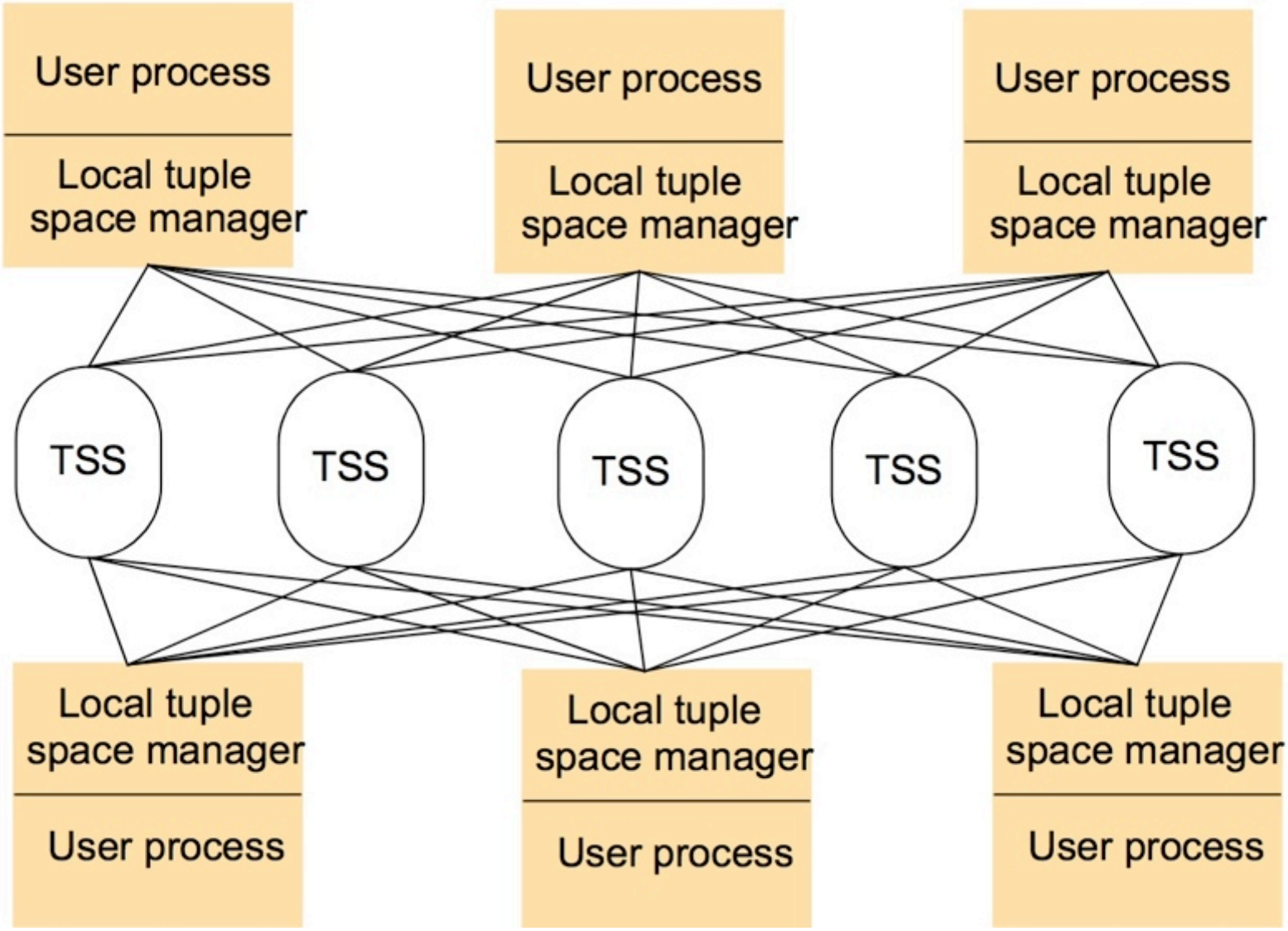
take *Phase 1: Selecting the tuple to be removed*

1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple

1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

Partitioning in the York Linda Kernel



The 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

Java class *AlarmTupleJS*

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

Java class *FireAlarmJS*

```
import net.jini.space.JavaSpace;
public class FireAlarmJS {
public void raise() {
      try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
       catch (Exception e) {
      }
   }
}
```

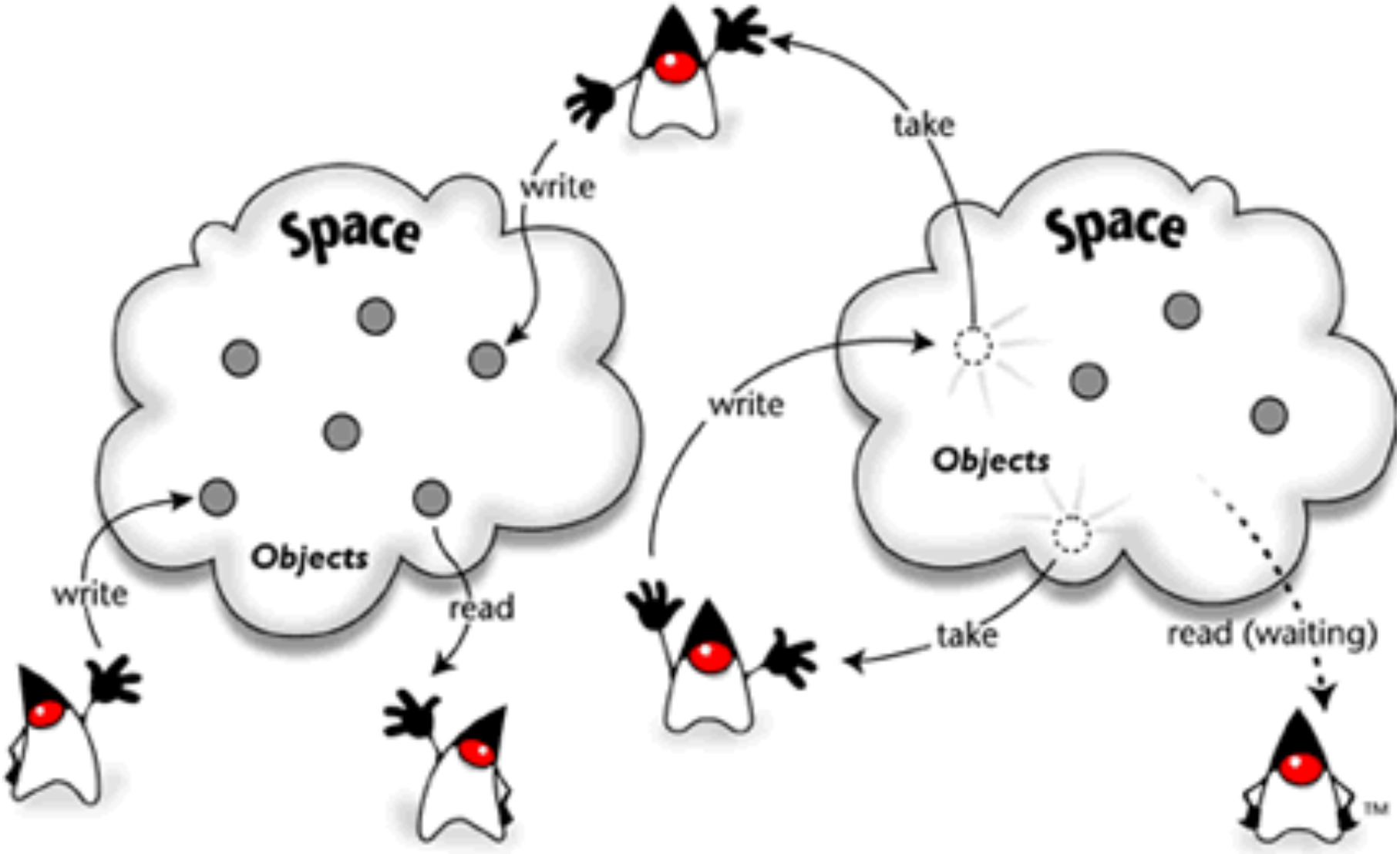
Java class *FireAlarmReceiverJS*

```
import net.jini.space.JavaSpace;
public class FireAlarmConsumerJS {
public String await() {
      try {
                  JavaSpace space = SpaceAccessor.findSpace();
                  AlarmTupleJS template = new AlarmTupleJS("Fire!");
                  AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                              Long.MAX_VALUE);
                  return recvd.alarmType;
      }
            catch (Exception e) {
                   return null;
                  }
         }
}
```

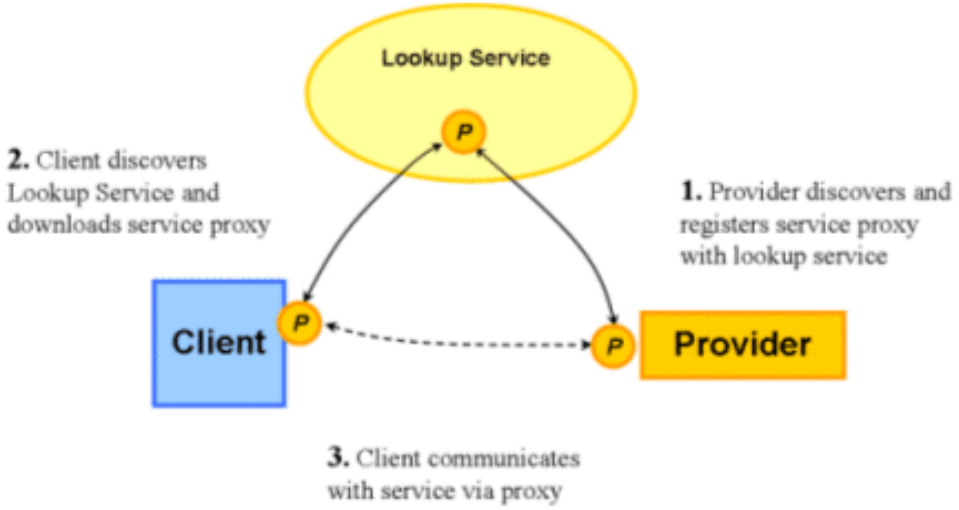
Jini (now Apache River)

- Jini is a software architecture supporting distributed systems as modular cooperating services
- River is the implementation of Jini service oriented architecture.
- It defines a programming model which both exploits and extends Java technology to enable the construction of secure, distributed systems consisting of federations of services and clients.
- Jini technology can be used to build adaptive network systems that are scalable, evolvable and flexible as typically required in dynamic computing environments

Jini supports JavaSpaces



Jini architecture



The diagram shows how a service provider registers a service with the Lookup Service, and how a client subsequently locates the service at the Lookup Service and begins working with the service

https://docs.gigaspaces.com/product_overview/about-jini.html

Jini / Apache River

- Jini uses a lookup service to broker communication between the client and service.
- Jini is based on a centralized model (though the communication between client and service can be seen as decentralized) that does not scale well to very large systems.
- the lookup service can be scaled by running multiple instances that listen to the same multicast group

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

Conclusions

- The three techniques: *groups*, *publish-subscribe*, and *message queues* offer a programming model that emphasizes communication (through messages or events), whereas *distributed shared memory* and *tuple spaces* offer a state-based abstraction.
- This is a fundamental difference in terms of scalability; the communication-based abstractions can scale to very large systems with an appropriate routing infrastructure

Exercise

- Discuss why indirect communication may be appropriate in volatile environments.
- To what extent can this be traced to time uncoupling, space uncoupling or indeed a combination of both?

Solution

- Volatile environments are environments where change is anticipated both in terms of the availability of computers and the underlying network.
- Mobile computing is one example of a highly volatile environment where users may connect and disconnect from the global network and may also experience periods of disconnection or weak connection because of location.
- Indirect communication is an appropriate strategy for such volatile environments for two key reasons:
  - Firstly, with the potential for failure, it is useful to have a communication mechanism that does not communicate directly with a given recipient, rather indirectly through an intermediary, providing a level of freedom for the system to replace, update, replicate or migrate the intended receiver (space uncoupling).
  - Secondly, with the potential for disconnection (of both sender and receiver), it is useful to have a communication mechanism that does not rely on both parties being available at the same time to communicate (time uncoupling).

Exercise

Message passing is both time- and space-coupled – that is, messages are both directed towards a particular entity and require the receiver to be present at the time of the message send.

Consider the case where messages are directed towards a name rather than an address and this name is resolved using DNS.

Does such a system exhibit the same level of indirection?

Solution

- in DNS, a name may map on to more than one IP address, for example to share load across a number of computers.
- Thus, a name server provides an additional level of indirection in that a sender may not be bound to a given receiver but rather refer to a logical name which is then bound to one of a number of possible receivers.
- This provides a similar effect to space uncoupling although the implementation details are different when compared to communicating through a more explicit intermediary such as a group, publish-subscribe system, message queue or shared memory abstraction