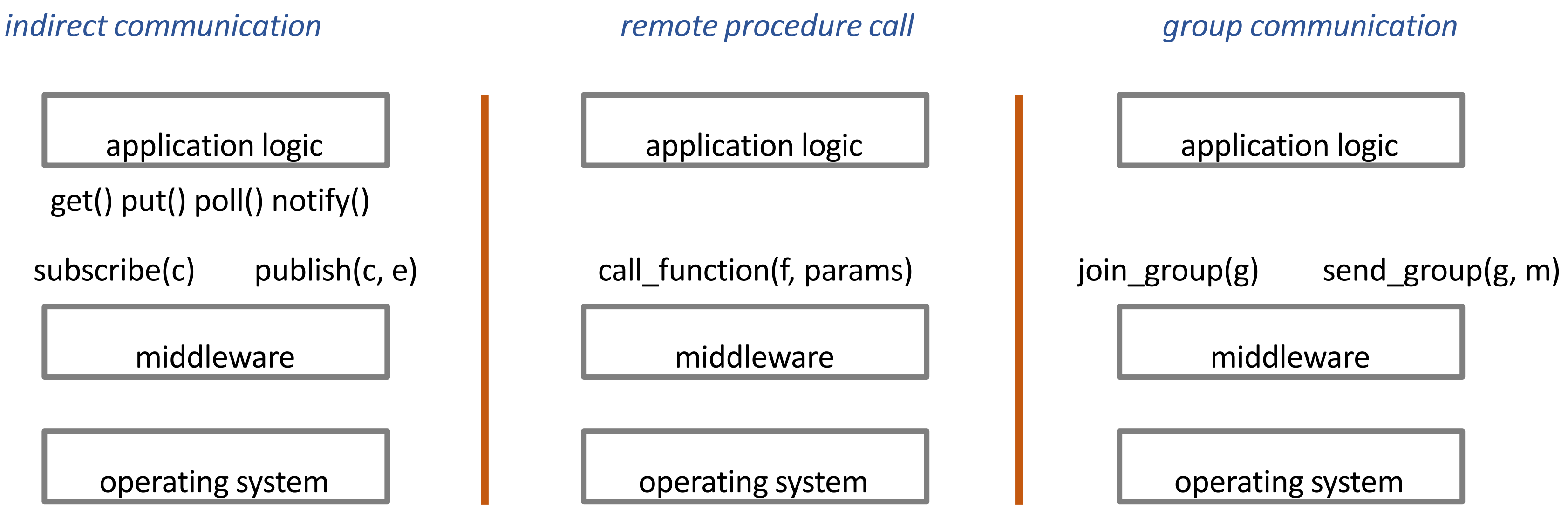
SCC311

Indirect Communication

Middleware

- Middleware is an abstraction, but usually one that has a complex implementation



What is Indirect Communication?

- Communication between entities in a distributed system through an abstraction with no direct coupling between the sender and the receiver(s)
  - → But what abstraction are we talking about?
    - Event-based abstractions (e.g. publish-subscribe)
    - Message queue abstraction
    - Shared memory abstractions (e.g. DSM, tuple spaces)
- Note the optional plural in this definition:
  - → Often intrinsically provides multiparty communication

→ Does Java RMI require direct coupling?

A Closer Look at Coupling

- **Remote invocation** paradigms all imply a direct coupling between the client and server
- Indirect communication paradigms seek loose coupling:
  - Space uncoupling in which the sender does not know or need to know the identity of the receiver(s) and vice versa
    - Because of this, the system developer has many degrees of freedom in dealing with change: participants (senders or receivers) can be replaced, updated, replicated or migrated
  - Time uncoupling in which the sender and receiver(s) can have independent lifetimes (in other words, the sender and receiver(s) do not need to exist at the same time to communicate)
    - Important benefits particularly in volatile environments where senders and receivers may come and go.
- Many uses in distributed systems including supporting mobility, dependability and event dissemination

On Indirection…

"All problems in computer science can be solved by another level of indirection"

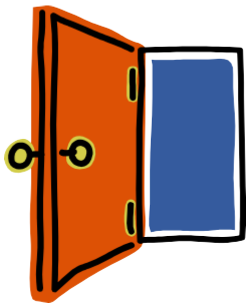
Roger Needham et al

"There is no performance problem that cannot be solved by eliminating a level of indirection"

Jim Gray

Group Communication:
An Initial Example of Indirection

- What is group communication?
  - Based on the concept of a group **abstraction**, with operations provided to join and leave the group (cf. group membership)
  - Messages sent to a group rather than to any individual process and messages are then delivered to each member of the group
  - Often enhanced by guarantees in terms of message ordering and reliability
- Uses in distributed systems
  - Important in supporting fault-tolerance
    - e.g. supporting replication
  - Also used heavily in dissemination of events
    - e.g. in financial systems
  - Examples: Jgroups, Akka



See lecture on group communication

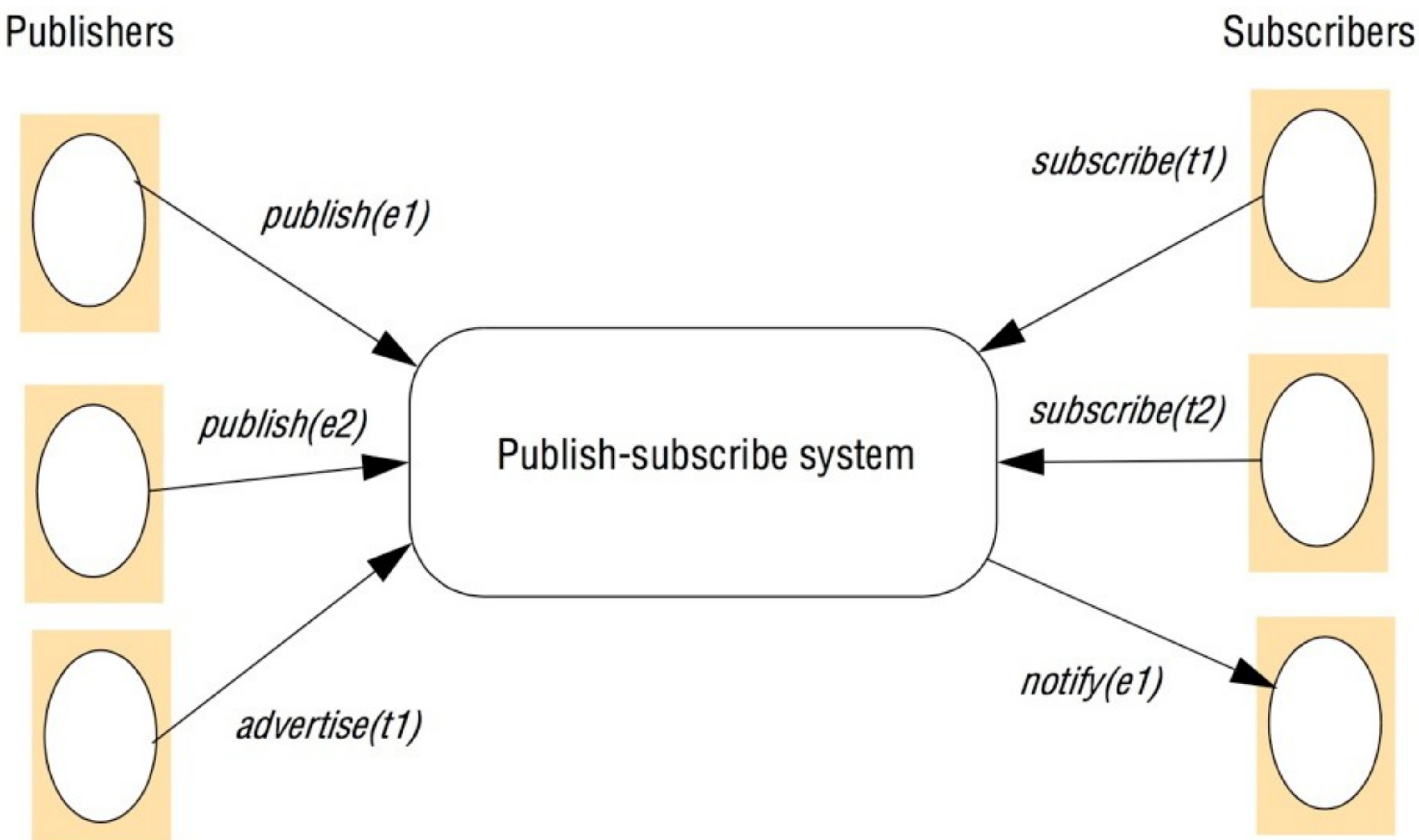
Publish-Subscribe Systems

- What is publish-subscribe (pub/sub) ?
  - A key example of a distributed event-based system whereby:
    - Publishers publish an event e: publish(e)
    - Subscribers express interest in a set of events specified by a **filter** f: subscribe(f)
    - Events are delivered asynchronously: notify(e)
    - Publishers optionally advertise what they will produce: advertise(f)
    - **The system acts as a broker to deliver events to the right subscribers**

- Uses in distributed systems
  - As with groups, used in financial information systems and related news feeds applications
  - Feature heavily in systems supporting cooperative working
  - Increasingly used in ubiquitous computing/ monitoring
  - Examples: Redis, Pulsar, Hermes, Scribe, Siena

Publish-Subscribe Systems

- The **clients** of the publish-subscribe system can be publishers, subscribers (or both in some systems).



Subscription Models

1. Channel-based

- → Publishers publish events to named channels and subscribers then subscribe to one of these named channels and therefore receive all events sent to that channel
- → Rather primitive scheme and the only one that defines a physical channel
- → All other schemes use some form of filtering over the content of an event as we will see below

Subscription Models (cont.)

2. Topic-based (also referred to as subject-based)

- ➔ Each notification is expressed in terms of a number of fields or attributes with one field denoting the topic and subscriptions defined in terms of this topic of interest
- ➔ Similar to channel-based approaches (implicit vs. explicit)
- ➔ Can be enhanced by introducing hierarchies of topics:
  - ➔ Example topics: “SCC/seminar”, “SCC/teaching”
  - ➔ Subscribing to a (parent) topic in the hierarchy (e.g., SCC) means subscribing to all the subtopics (seminars and teaching) of the parent
- ➔ Not possible to filter events within a topic without creating (and subscribing to) a specialised subtopic.

Subscription Models (cont.)

3. Type-based

- Intrinsically linked with (typed) object-based approaches
- Subscriptions defined in terms of type, with matching defined in terms of types or subtypes of the given filter
- Can be **integrated elegantly into programming languages**
- Can have types belonging to multiple supertypes
  - Can be more flexible than the topic hierarchy
- Not possible to filter events within a type without creating (and subscribing to) a specialised subtype.

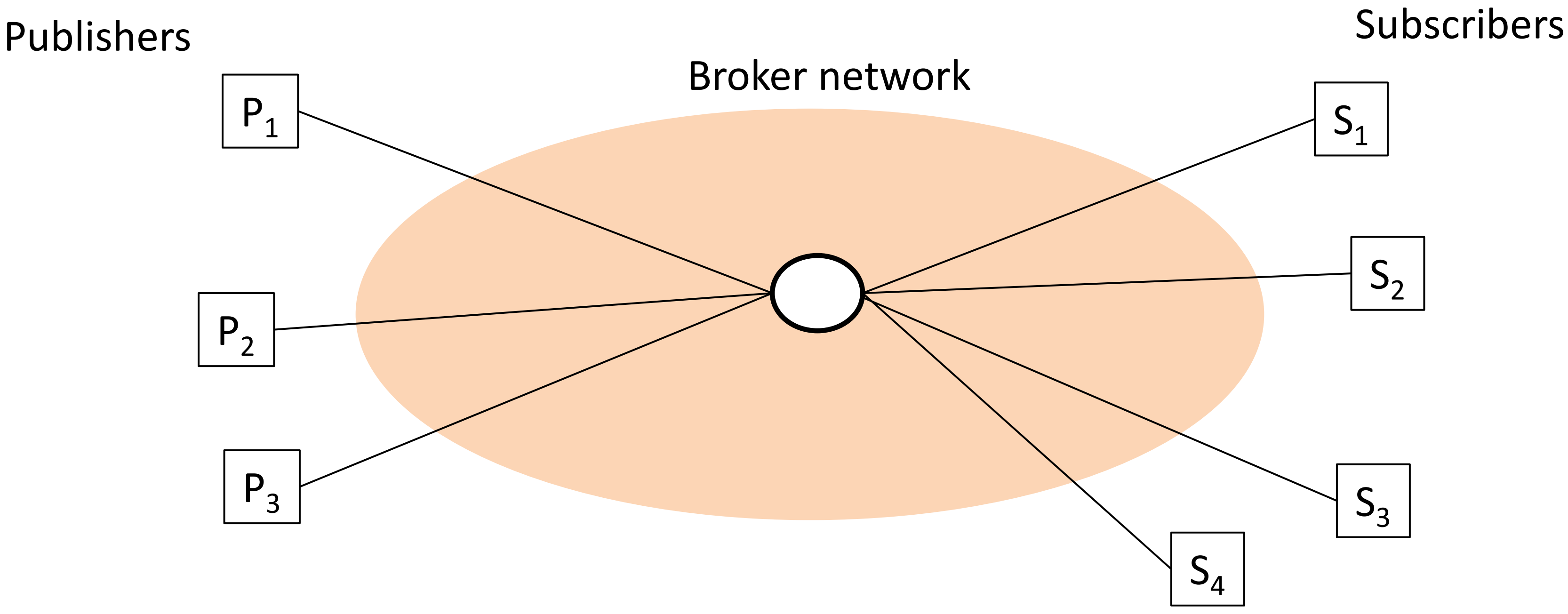
Subscription Models (cont.)

4. Content-based

- Generalization of static classification schemes based on topics or types,
  - based on allowing the expression of subscriptions over a range of fields in an event notification
- The *filter* is a query defined in terms of compositions of constraints over the values of event attributes
- Significantly *more expressive* but with significant new challenges introduced in terms of implementation
  - A query language is used in these systems to express constraints

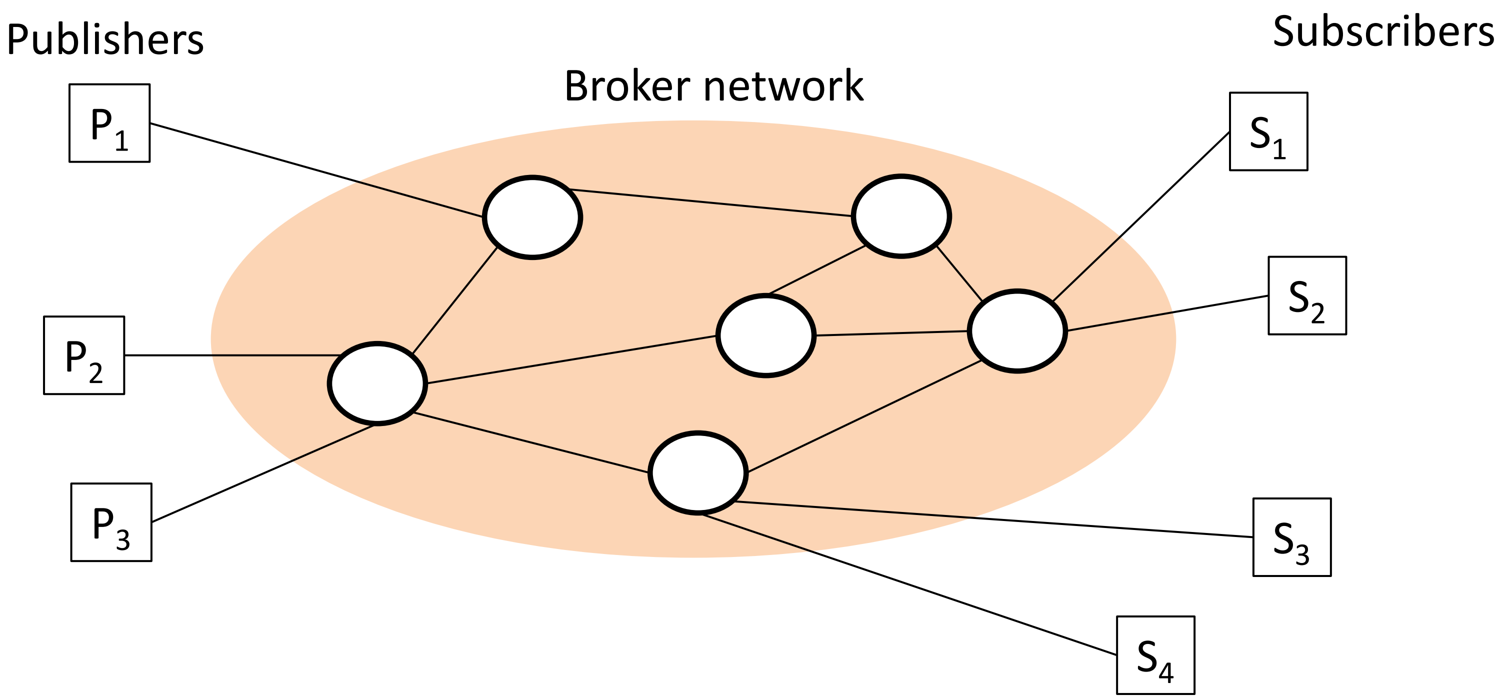
Implementing PubSub

- The key decision is whether to go for a centralised, distributed or fully peer-to-peer architecture
- A **centralised architecture** with one single broker (server) is not scalable or resilient



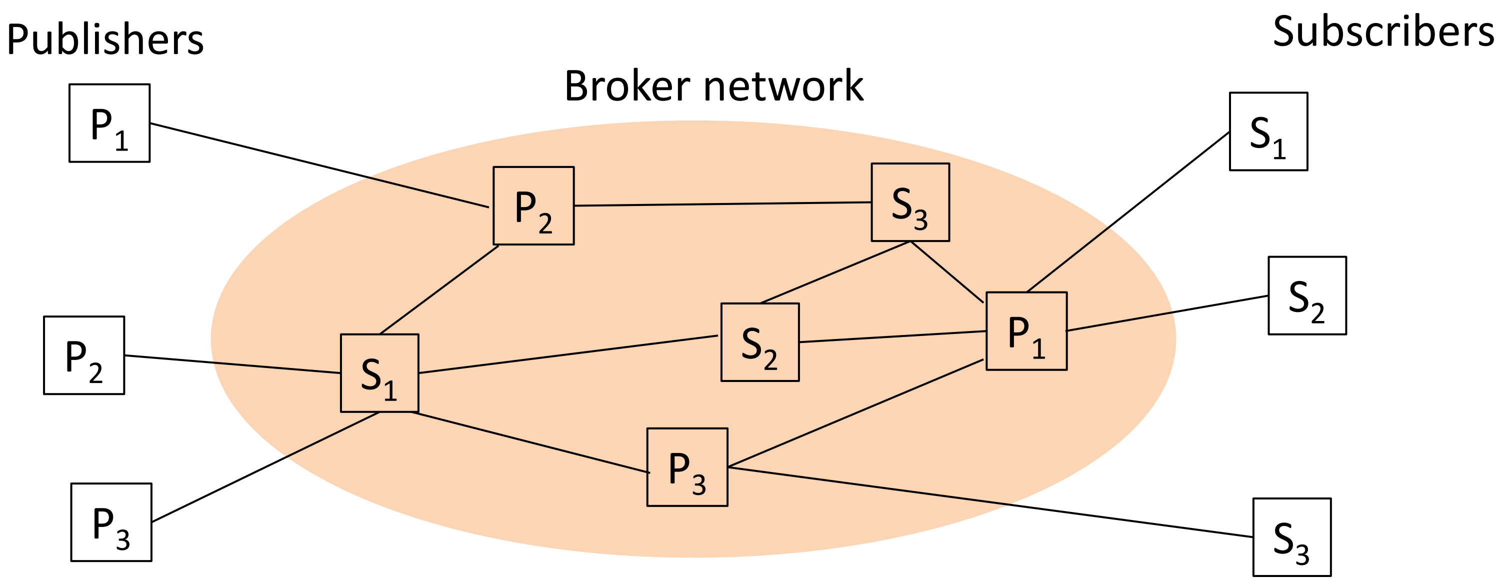
Implementing PubSub

- The key decision is whether to go for a centralised, distributed or fully peer-to-peer architecture
- Most large-scale systems have **distributed architectures** consisting of a broker network with multiple brokers



Implementing PubSub

- The key decision is whether to go for a centralised, distributed or fully peer-to-peer architecture
- In a peer-to-peer architecture all clients are also brokers themselves
  - No dedicated broker server(s)



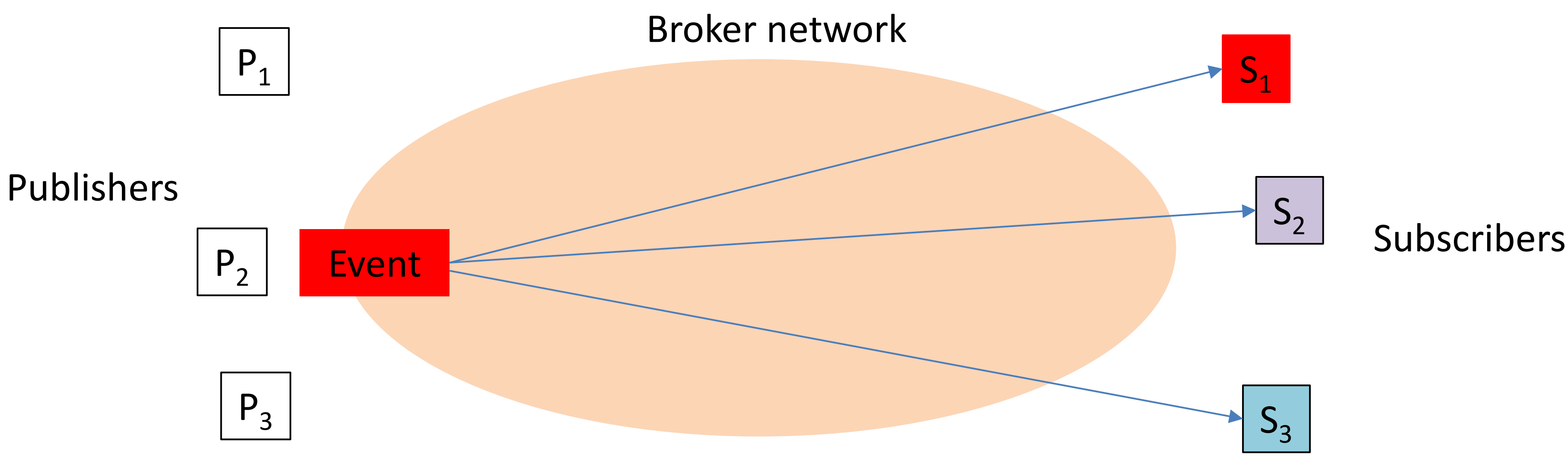
Implementing a PubSub System

- **Main challenge**: routing of event notifications (sent by publishers) as efficiently as possible to appropriate subscribers
- The distributed implementations of channel-based, topic-based, and type-based schemes are relatively easy:
  - The number of groups (event types, topics or channels) are known at any given time
  - Simple mapping of each channel or topic onto associated group
    - Use group communication (e.g. IP multicast) if available to send events to groups
- The distributed implementation of content-based approaches is more complex and deserves further consideration.
  - The group for a given event must be computed at “run-time” (applying filters that express constraints)

Implementing a Distributed Content-based PubSub System

1. Flooding of events to the subscribers

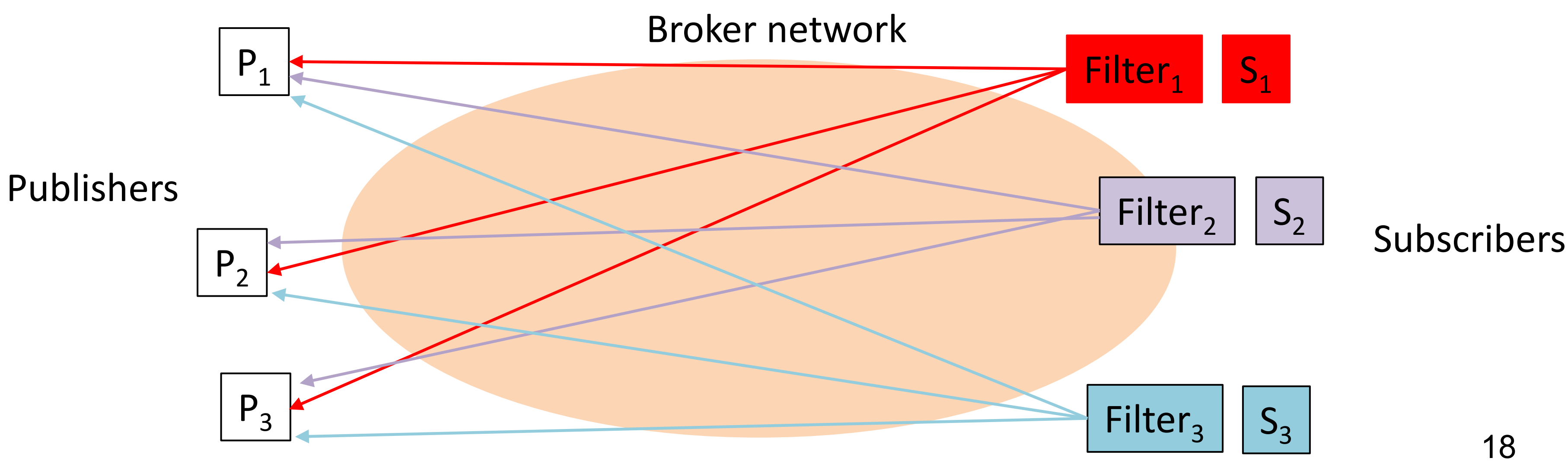
- ➔ Send all published events to all possible recipients
- ➔ Perform matching of filters at the subscribers
- ➔ Simple but lots of unnecessary event transfers!
  - ➔ matching of events against filters should take place as close to the source (publishers) to avoid unnecessary event transfers.



Implementing a Distributed Content-based PubSub System

2. Flooding of filters to publishers

- Filters are propagated all the way to publishers
- Then all publishers will know all the subscribers (defeats the purpose of indirection)



Implementing a Distributed Content-based PubSub System

3. Filtering-based routing

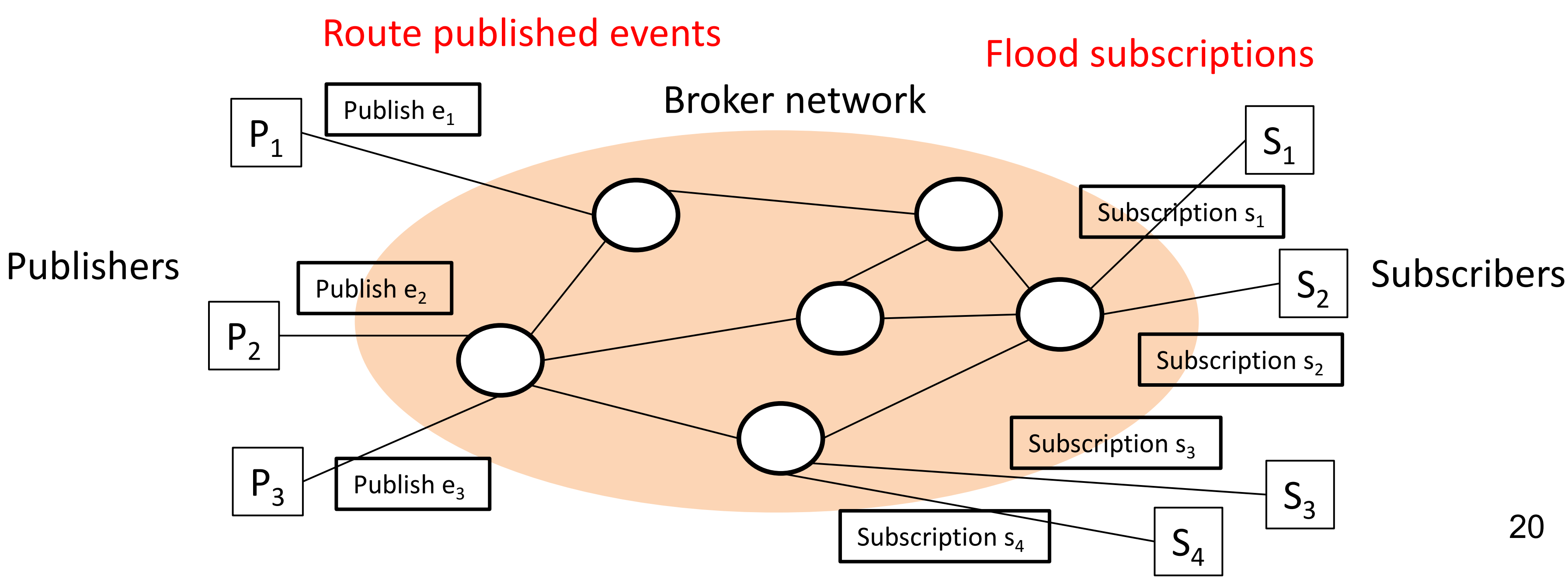
- Filters are propagated back (flooded) through the broker network towards all publishers
  - store associated filter state at each broker (to form a routing table)
- Matching against filters is done on each node in the broker network
- Brokers maintain the following state:
  - **Subscription list** of all subscribers that are directly connected
  - **A routing table** to match events to neighbouring brokers and directly-connected subscribers
- A notification should ideally be forwarded through the broker network along a path only if that path leads to a valid (matching) subscriber

Implementing a Distributed Content-based PubSub System

3. 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;
```



Implementing a Distributed Content-based PubSub System

3. Advertisement-based

- ➔ Advertisements are propagated (flooded) through the broker network towards all subscribers
  - ➔ store associated advertisement state at each broker
- ➔ Subscriptions are then used to establish paths:
  - ➔ Propagate subscriptions (towards the publishers) following paths with advertisements containing matching fields
  - ➔ Form a routing table which only consists of a subset of subscriptions
  - ➔ All advertisements are stored at each node

➔ In the filter-based approach all subscriptions are stored at all brokers vs in the advertisement-based all advertisements are stored at all brokers

Implementing a Distributed Content-based PubSub System

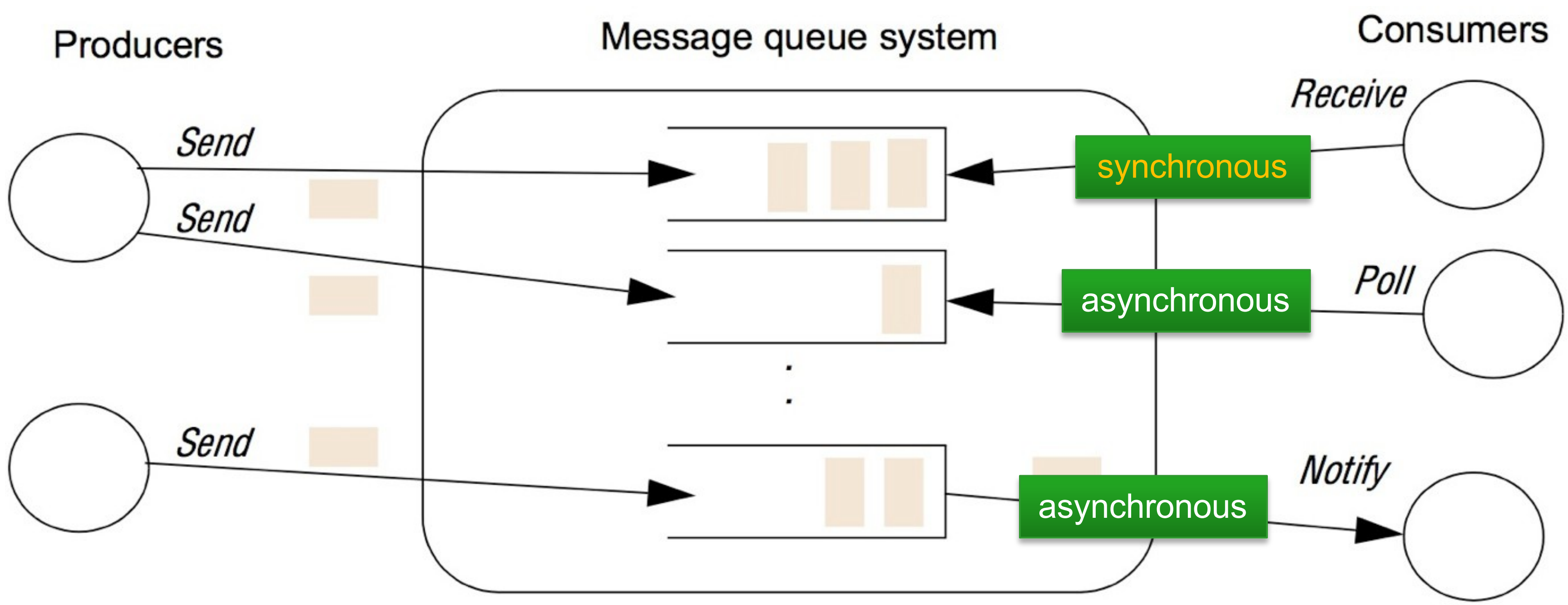
4. Rendezvous

- Consider the set of all possible events as an event space
- Partition this event space into pieces and allocate responsibility for each piece to a given broker (known as the rendezvous node for that event)
- Implementation requires two functions to be defined:
  - SN(s) which takes a given subscription, s, and returns one or more rendezvous nodes which take responsibility for that subscription
  - EN(e) which takes a given event, e, and returns one or more rendezvous nodes responsible for matching e against subscriptions in the system
- Can lead to highly scalable implementations
- A Distributed Hash Table (DHT) can be used to implement rendezvous

Message Queues

- What is a message queue?
  - ➔ An alternative paradigm for indirect communication based on distributed queues
    - Messages are sent to a queue
    - Processes can then access messages in the queue either by receiving a message (blocking), polling for messages (non-blocking) or being notified when messages arrive
    - Messages are **persistent and message delivery is reliable**
    - Fundamentally a point-to-point service (not multi-party)
- Uses in distributed systems
  - ➔ Enterprise Application Integration (EAI), i.e. integration between applications in a given enterprise utilising the loose coupling
  - ➔ Also used heavily in commercial transaction processing systems
  - ➔ Examples: JMS, IBM Websphere MQ, RabbitMQ, ZeroMQ, Microsoft MSMQ, Oracle Streams Advanced Queuing

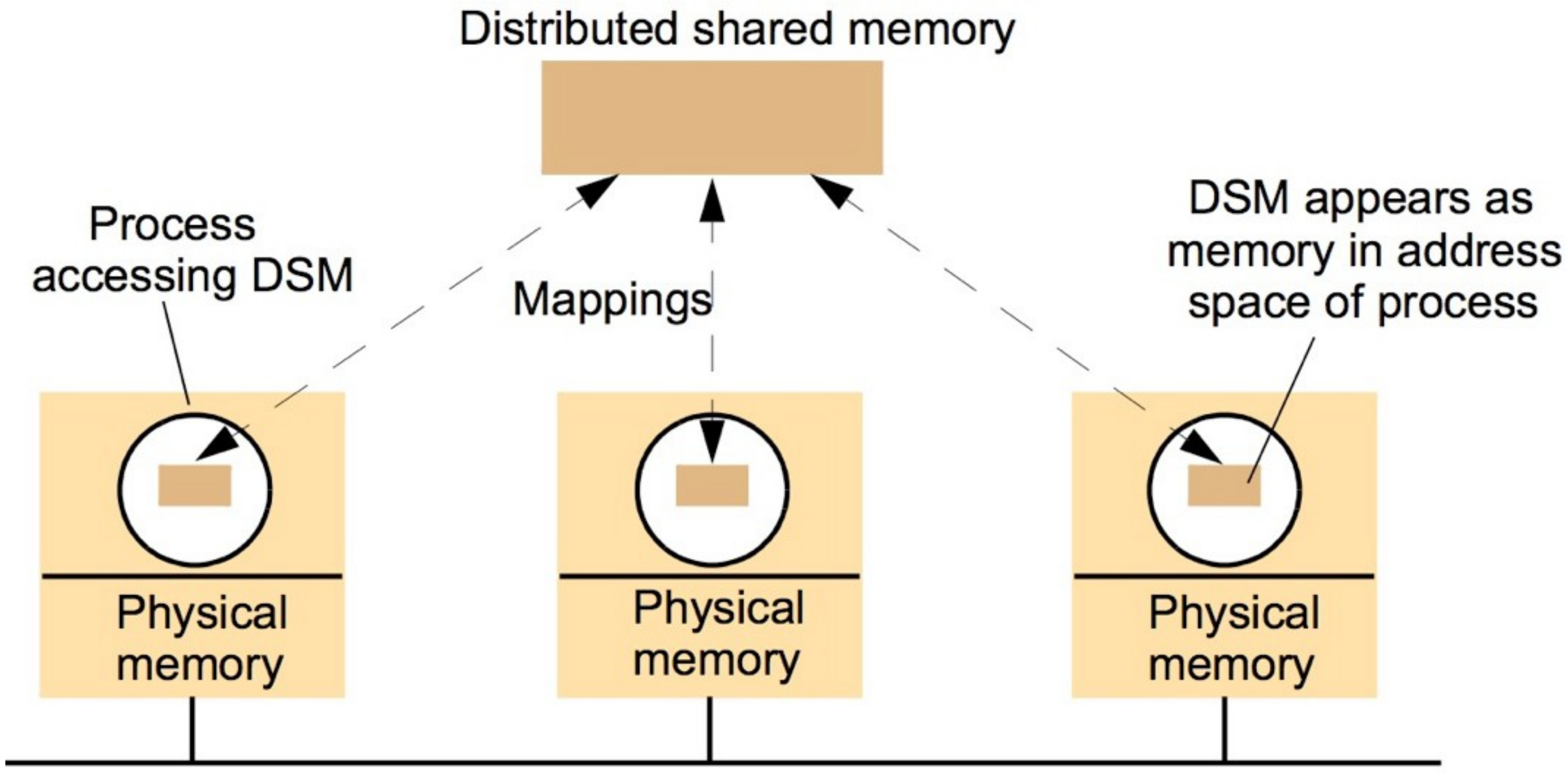
Message Queues (continued)



Distributed Shared Memory

- What is a distributed shared memory?
  - Provides an abstraction of shared memory in a distributed system
  - If data is not available locally, it must be fetched (similar to a page fault in traditional virtual memory systems)
  - Hides distribution entirely from the programmer
  - No new programming abstractions to learn => portable
  - Can be costly to implement:
    - maintaining consistency of shared data (through atomicity)
    - preventing race conditions
- Uses in distributed systems
  - Tends to be more specialist, for example for parallel and distributed computation in cluster computers (i.e. relatively tightly coupled distributed architectures)
  - Examples: OpenSSI, MOSIX, DIPC

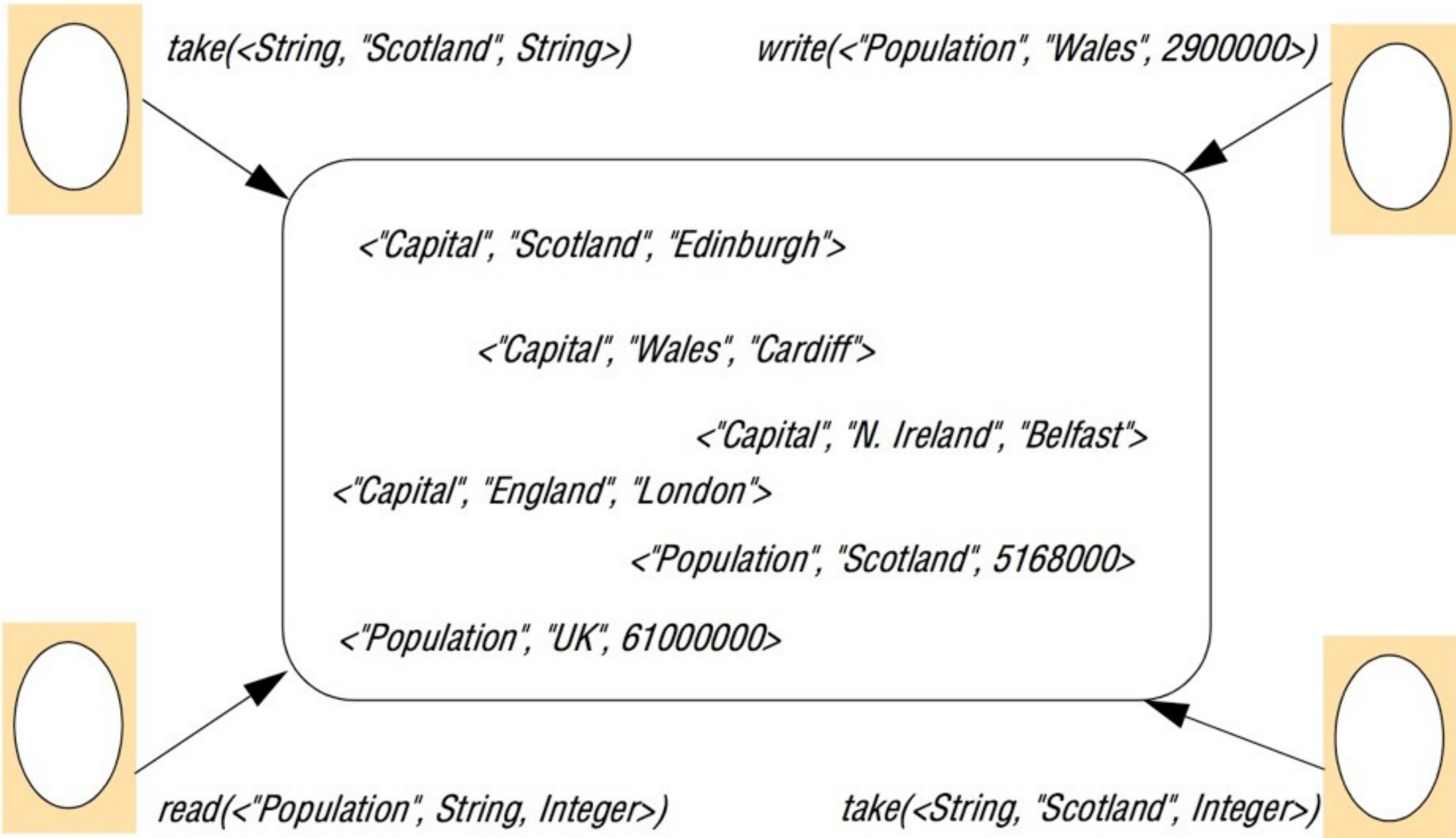
Distributed Shared Memory (continued)



Tuple Spaces

- What is a tuple space?
  - Another paradigm for indirect communication offering an abstraction of a semi-structured shared space consisting of a number of tuples
    - Processes can write tuples to the tuple space
    - Processes can then read a tuple from tuple space which also leaves a copy in the tuple space
    - Alternatively, they can take the tuple which removes it from the space
    - Both the read and take operations are based on pattern matching (associative access)
- Uses in distributed systems
  - Influential in the areas of mobile and ubiquitous computing
  - Also used for systems integration
  - Examples: Linda, JavaSpaces, IBM TSpaces

Tuple Spaces (continued)



Comparison of Approaches

	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

Expected Learning Outcomes

At the end of this session:

- You should understand the essence of indirect communication, why it is different from remote invocation (for example) and the role of time and space uncoupling in supporting these characteristics

- You should appreciate the range of indirect communication services available and be able to assess their strengths and weaknesses

- You should have a brief understanding of group communication as a service and how it relates to other indirect paradigms

- You should have a more in-depth understanding of the publish-subscribe approach including the various subscription models, why this choice is crucial and also how they may be implemented in a distributed environment

- You should be able to compare and contrast publish-subscribe with message queues and also the shared memory abstractions offered by distributed shared memory or tuple spaces

Associated Reading

- CDKB, chapter 6