COMP90015 Distributed Systems
Semester 2, 2023
Topic: Indirect Communication

Dr Tawfiq Islam
School of Computing and Information Systems (CIS)
The University of Melbourne, Australia

Learning Outcomes

- Space Uncoupling and Time Uncoupling
- Group Communication
- Publish Subscribe Systems
- Distributed Message Queues
- Distributed Shared Memory
- Tuple Spaces

- Reading: Distributed Systems: Concepts and Design by George Coulouris (5th edition). Chapter 6. Sections: 6.1, 6.2, 6.3, 6.4, 6.5

Space and Time Uncoupling in Distributed Systems

- **Space uncoupling** – sender does not know or need to know the identity of the receiver(s)
- **Time uncoupling** – sender and receiver can have independent lifetimes; they do not need to exist at the same time

	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
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

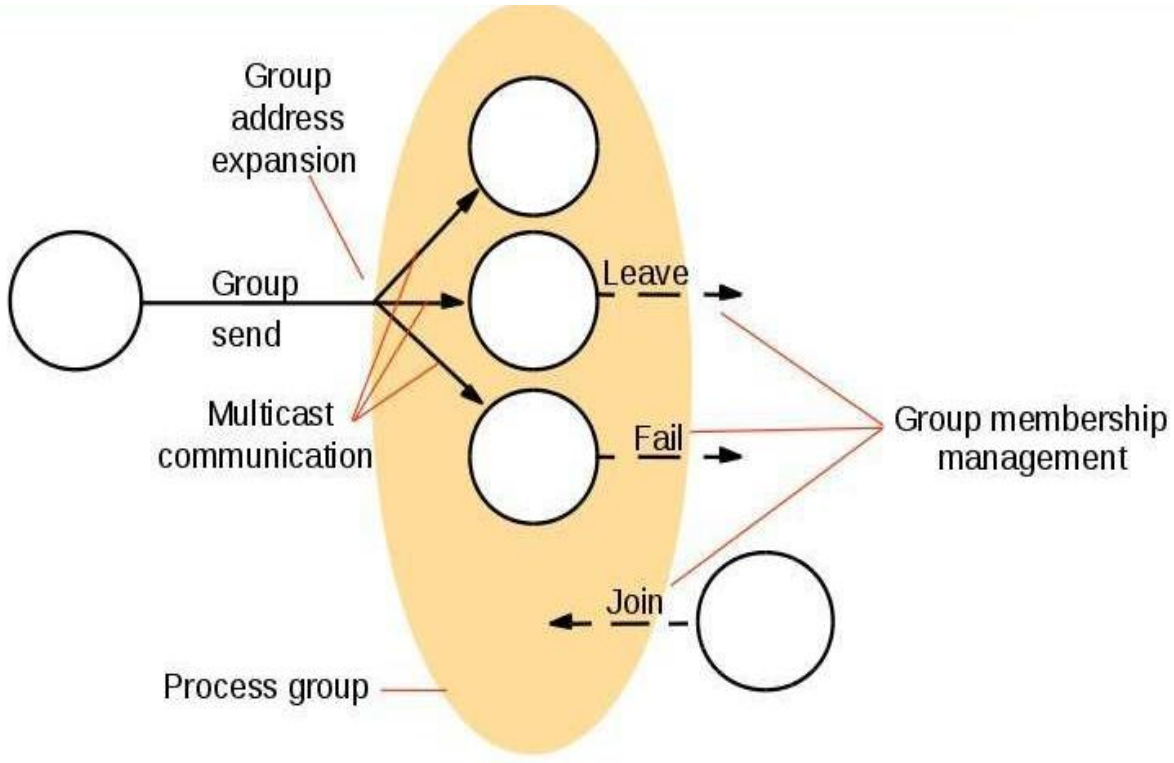
Indirect Communication

- Direct communication is communication that takes place directly between the communicating processes, whereas **indirect communication** is defined as communication between entities in a distributed system through an **intermediary** with **no direct coupling between** the sender and the receiver(s).
- **Time uncoupling** is **not** synonymous with asynchronous communication. Asynchronous communication doesn't imply that the receiver has an independent lifetime, in other words we could consider a time coupled asynchronous system.
- Indirect communication paradigms tend to be described in terms of a **metaphor** that aids in understanding the expectations of the paradigm and for what kinds of distributed applications it is useful.

Group Communication

- **Group communication** offers a **space uncoupled** service whereby a message is sent to a group and then this message is delivered to all members of the group.
- It provides **more** than a primitive IP multicast:
  - manages group membership
  - detects failures and provides reliability and ordering guarantees
- Efficient sending to multiple receivers, instead of multiple independent send operations, is an essential feature of group communication.
- Example: JGroups Toolkit

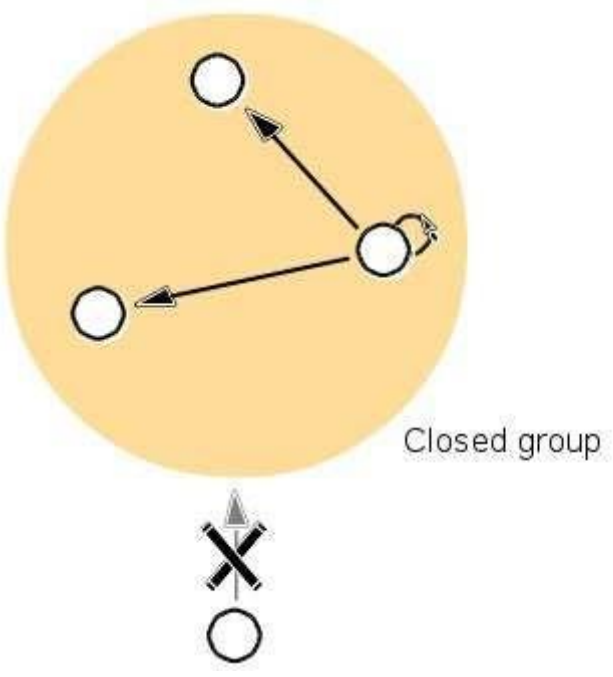
Group Communication Example

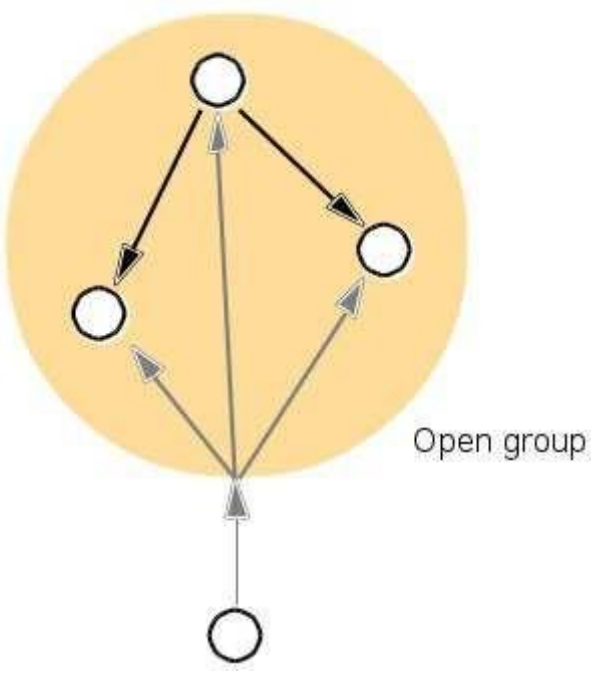


Typical aspects of a **Group API**:

- group creation
  - create/delete a group
  - list/search available groups
- group membership
  - join/leave a group
  - list members of a group
- ***multicast*** to selected members of a group, ***broadcast*** to all members

Closed Group vs Open Group





- Closed groups only allow group members to multicast to it
- Overlapping groups allows entities to be members of multiple groups
- Synchronous and asynchronous variations can be considered

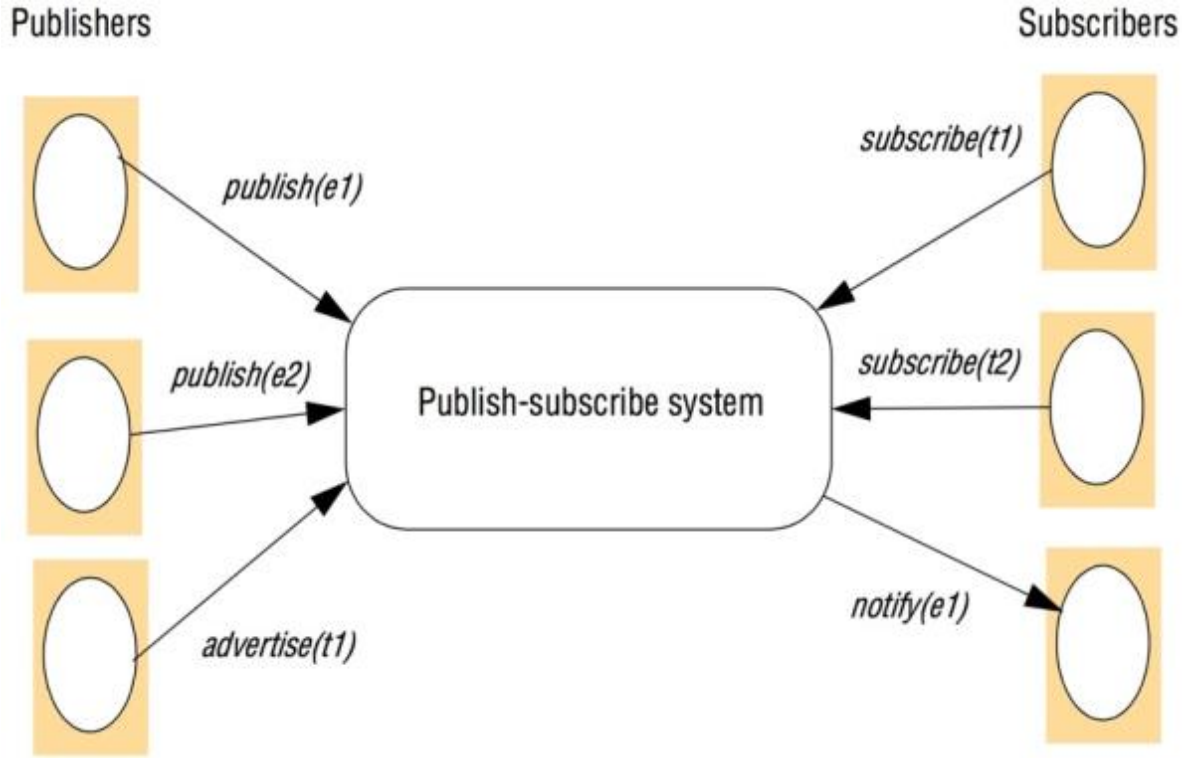
Implementation Issues

- Reliability and ordering in multicast
  - **FIFO (first in first out)** ordering is concerned with preserving the order from the perspective of a sender process
  - In **causal ordering**, a message that ***happens before*** another message will be preserved in that order in the delivery at all processes
  - In **total ordering**, if a message is delivered before another message at one process, then this is preserved at all processes
- Group membership management
  - group members leave and join
  - failed members
  - notifying members of group membership changes
  - changes to the group address

Publish-Subscribe Systems

- **Publish/subscribe** systems are sometimes referred to as **distributed event-based systems**. A publish/subscribe system is a system where **publishers** (event sources) publish structured **events** to an **event service** and **subscribers** express interest in particular events through **subscriptions** which can be arbitrary patterns or query expressions over the structured events.
  - financial information systems
  - live feeds of real-time data, e.g., RSS feeds
  - support for cooperative working, where a number of participants need to be informed of updates
  - support for ubiquitous computing, including management of events emanating from the ubiquitous infrastructure, e.g., location events
  - a broad set of monitoring applications, including network monitoring on the Internet

Programming Model



- publish(e) // e is an event
- subscribe (f) // f is a filter (a pattern defined over set of events)
- unsubscribe(f)
- notify(e) // The system notifies a subscriber of an event
- Some systems have advertise(f) to declare the nature of future events
- Revoke advertisement with unadvertise(f)

Types of Publish-Subscribe Systems

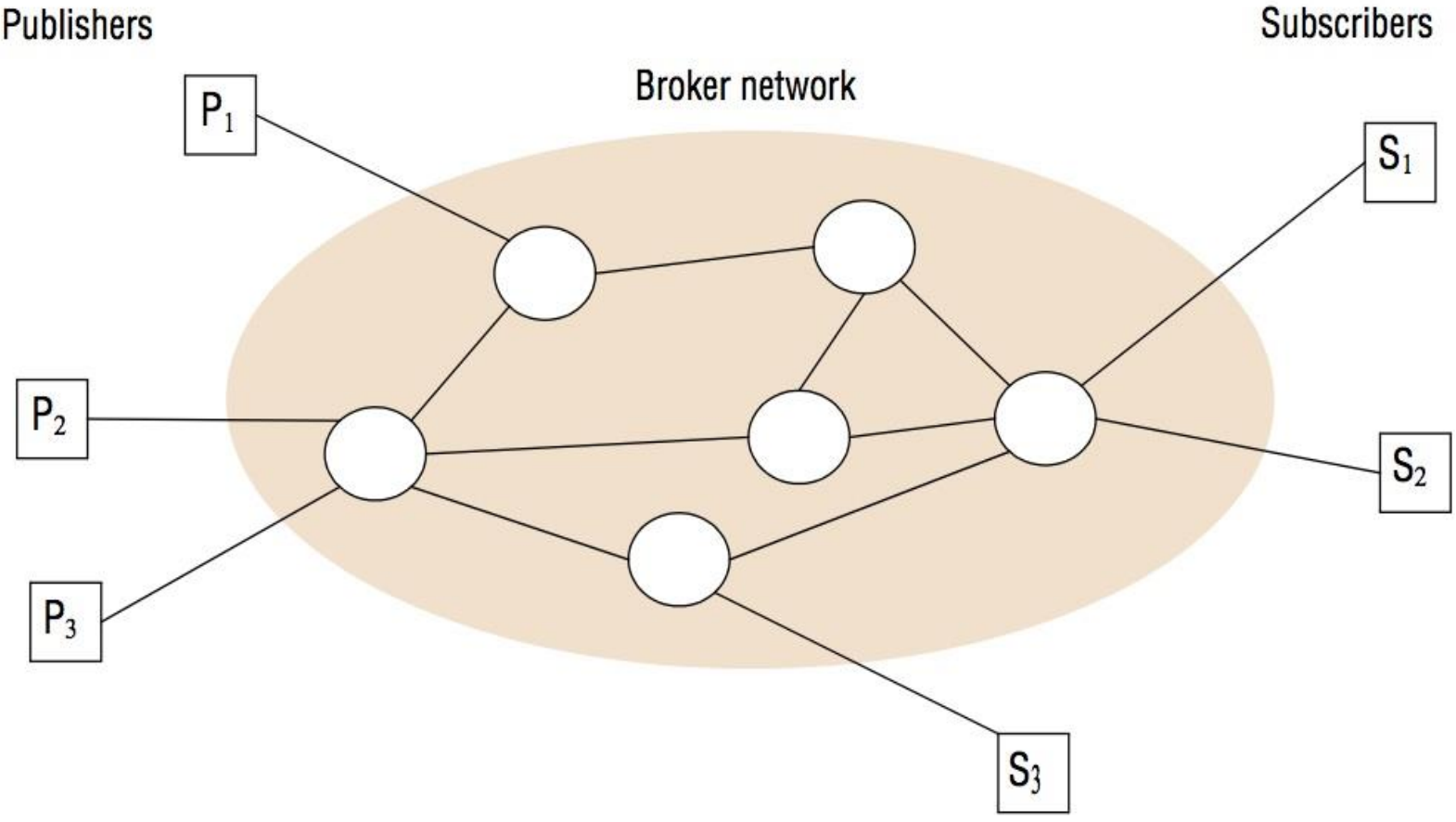
- **Channel Based** – Publishers publish to named channels and subscribers subscribe to all.
- **Topic Based** – Subscribers register interest in particular topics and notifications occur when any information related to the topic arrives.
- **Content Based** – This is the most flexible of the schemes. Subscribers can specify interest is particular values or ranges of values for multiple attributes. Notifications are based on matching the attribute specification criteria.
- **Type Based** – Subscribers register interest in types of events and notifications occur when particular types of events occur.

When and event matches a subscriber's subscription then the system sends a **notification** that contains the event to the subscriber.

- **Implementation:**
  - Centralized: single server node as an event broker
  - Distributed Broker Network: survives node failures
  - Peer to Peer (P2P): No distinction between publishers, subscribers, and brokers.

Distributed Broker Network

The ***Broker*** exchanges or routes information from publishers to subscribers.

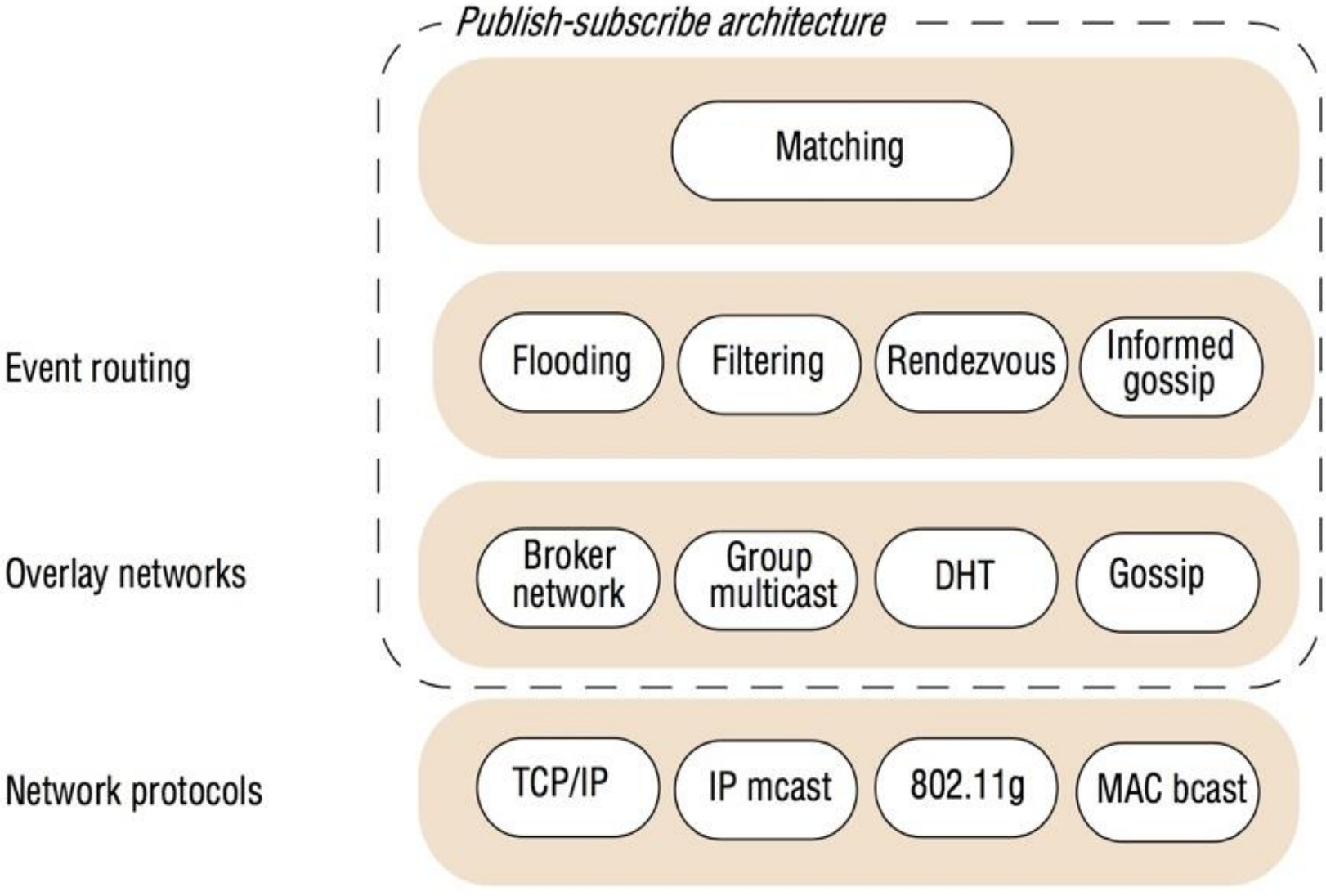


Characteristics of Publish-Subscribe Systems

- Heterogeneity
  - When event notifications are used as means of communication, components that are not designed to work together, can now work together.
- Asynchronicity
  - Notifications are sent **asynchronously** by event-gathering publisher to all subscribers that expressed interest in them.
  - Publishers do not need to synchronize with subscribers.
- Delivery of Notifications
  - Reliability: Is delivery of a notification guaranteed?
    - ✓ Some applications can tolerate loss of notifications.
    - ✓ Some applications require all notifications be delivered (e.g., for the sake of fairness).
  - Timing: How fast should a notification be delivered?
    - ✓ Real-time, or
    - ✓ Delayed delivery is acceptable.

13

Publish-Subscribe Architecture

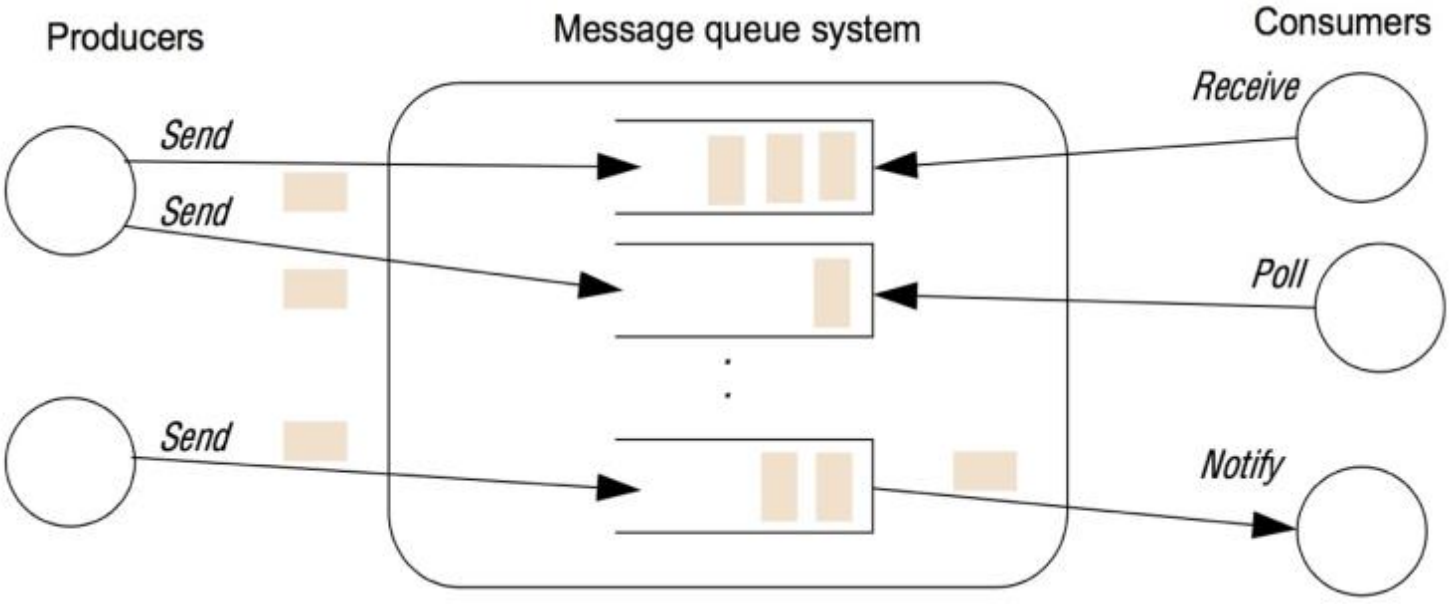


Example Publish-subscribe Systems

A modern example of a pub/sub system is **Apache Kafka**. Others are shown below.

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

Distributed Message Queues

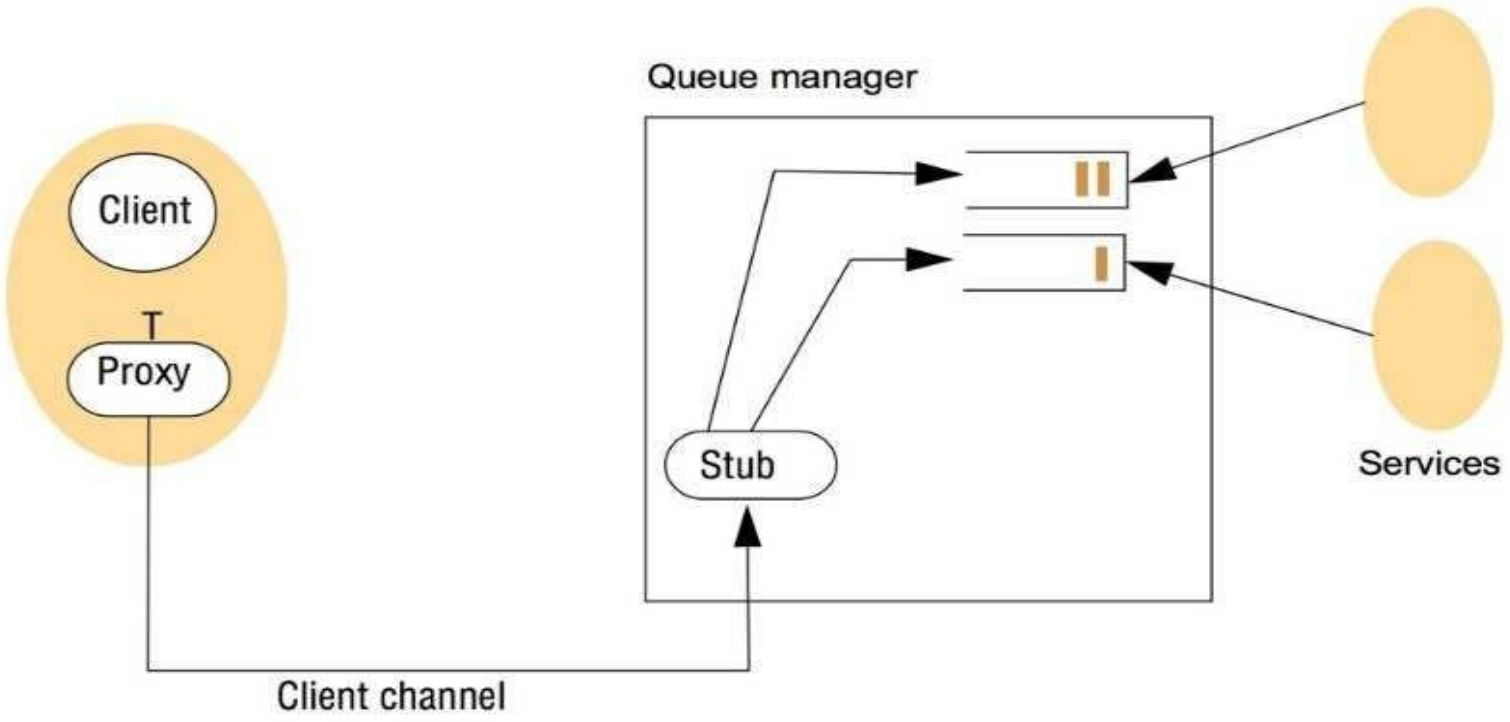


- Message queues provide a point-to-point service using the concepts of a **message** for data encapsulation and **queue** as an indirection. They are point-to-point in that each message is sent by a single process – **producer** – and received by a single process – **consumer**.
- Since communication uses messages, the message queue paradigm may not be suitable for applications that require streaming data or bulk data transfer.
- Good for distributing units of work to processes and command/control type operations.

Programming Model

- Usually, the message queue system is expected to provide reliability in that messages are not dropped or lost, and since it's a queue, messages are received in the order sent. The API is very much the same as a blocking queue that is used in concurrent programming.
  - **send** – producers put a message on a particular queue, may block the sender if the queue has finite capacity.
  - **blocking receive** – a consumer waits for at least one message on a queue and then returns.
  - **non-blocking receive** – or **poll**, a consumer will check and get a message if there, otherwise it returns without a message.
  - **notify** – a signal is sent to the consumer when messages are available on the queue for consumption.
- It is useful to consider this API in terms of actual processes and low-level exchange protocols. E.g. the implementation may use TCP for producers and consumers to connect to the queueing system.

Case Study: IBM WebSphereMQ



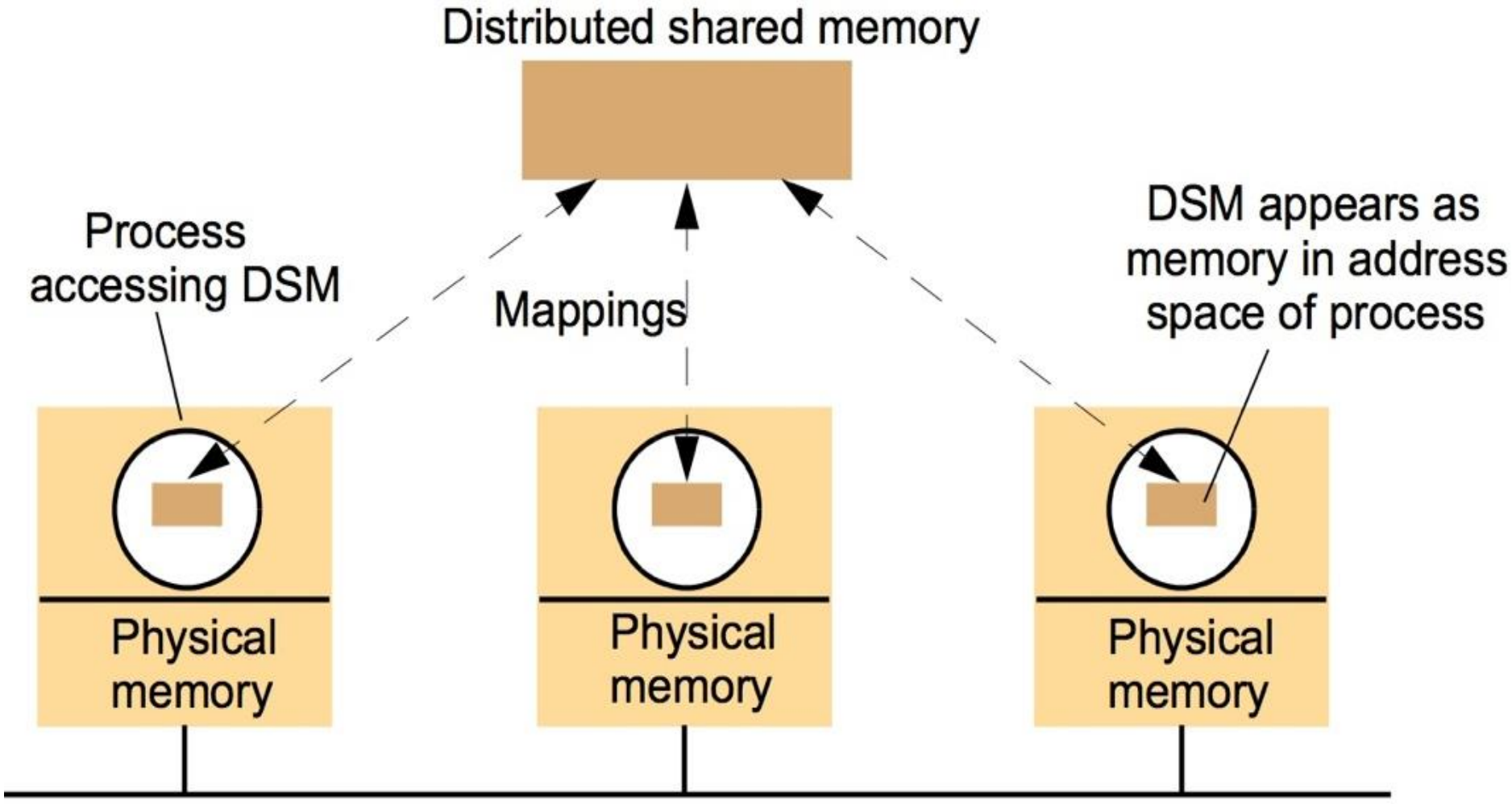
- A message queueing system typically provides a library for the programming to build a client, either a producer or a consumer, and a server implementation that implements the queue manager itself. The server implementation will typically run a process that allows producers and consumers to connect.

Distributed Shared Memory (DSM)

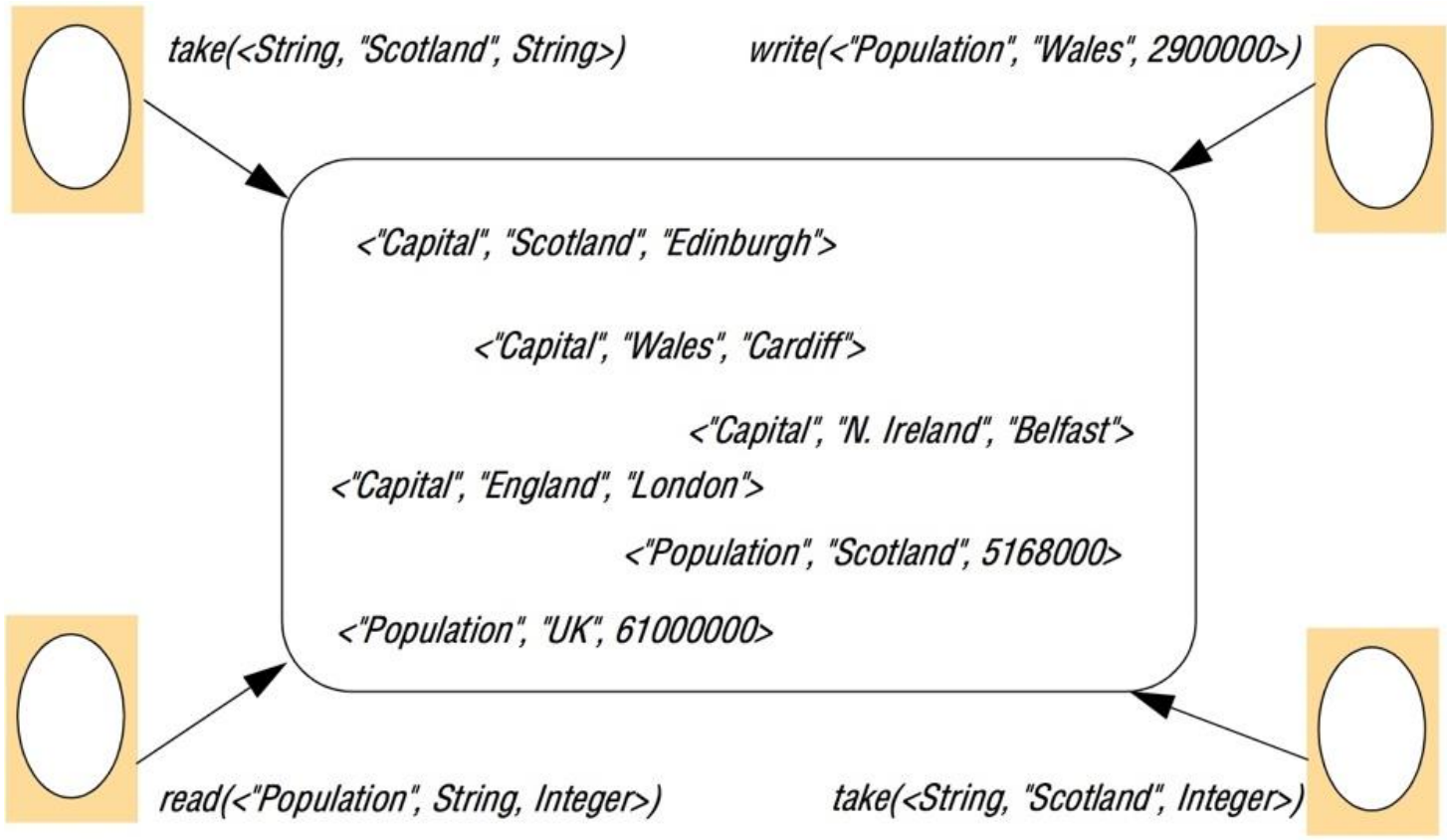
- **Distributed Shared Memory (DSM)** spares the programmer the concerns of message passing.
- DSM is a tool for parallel, distributed, or applications where **individual shared data items** can be accessed **directly**.
- DSM runtime has to send updates in messages between computers.
- Example of DSM is the Apollo Domain File System
  - Processes on different hosts share files by mapping them directly onto their address space.
  - Persistent DSM
- DSM became significant with the growth of shared-memory multiprocessors.

19

DSM Abstraction



Tuple Space



- Processes communicate directly by placing **tuples** in a **tuple space**, from which other processes can read or remove them.
- Tuples do not have an address but are accessed by **pattern matching** on content
- Example: JavaSpaces

Programming Model

- Tuples consist of a sequence of one or more **typed** data fields, e.g., <“fred”, 1958>, <4, 9.8, “Yes”>.
- Any combination of types of tuples may exist in the same tuple space.
- Processes share data by accessing the same tuple space.
- Operations
  - **write**: place a tuple in the tuple space.
  - **read**: return the value of a tuple
  - **take**: return and remove the tuple.
  - The read and take operations both block until there is a matching tuple in the tuple space. This is a form of **synchronization**.

22

Programming Model (cont.)

- Example operations
  - take(<String, integer>): This can extract <“fred”, 1958>, <“sid”, 1964>.
  - take (<String, 1958>) only extracts the tuple for fred only.
- Tuples are **Immutable** (can not be accessed directly) and have to be replaced in the case of a modification.
  - To implement a counter <“counter”, 64> in order to increment the counter:
    - `<s, count> := myTS.take (<“counter”, integer>);`
    - `myTS.write (<“counter”, count + 1>);`

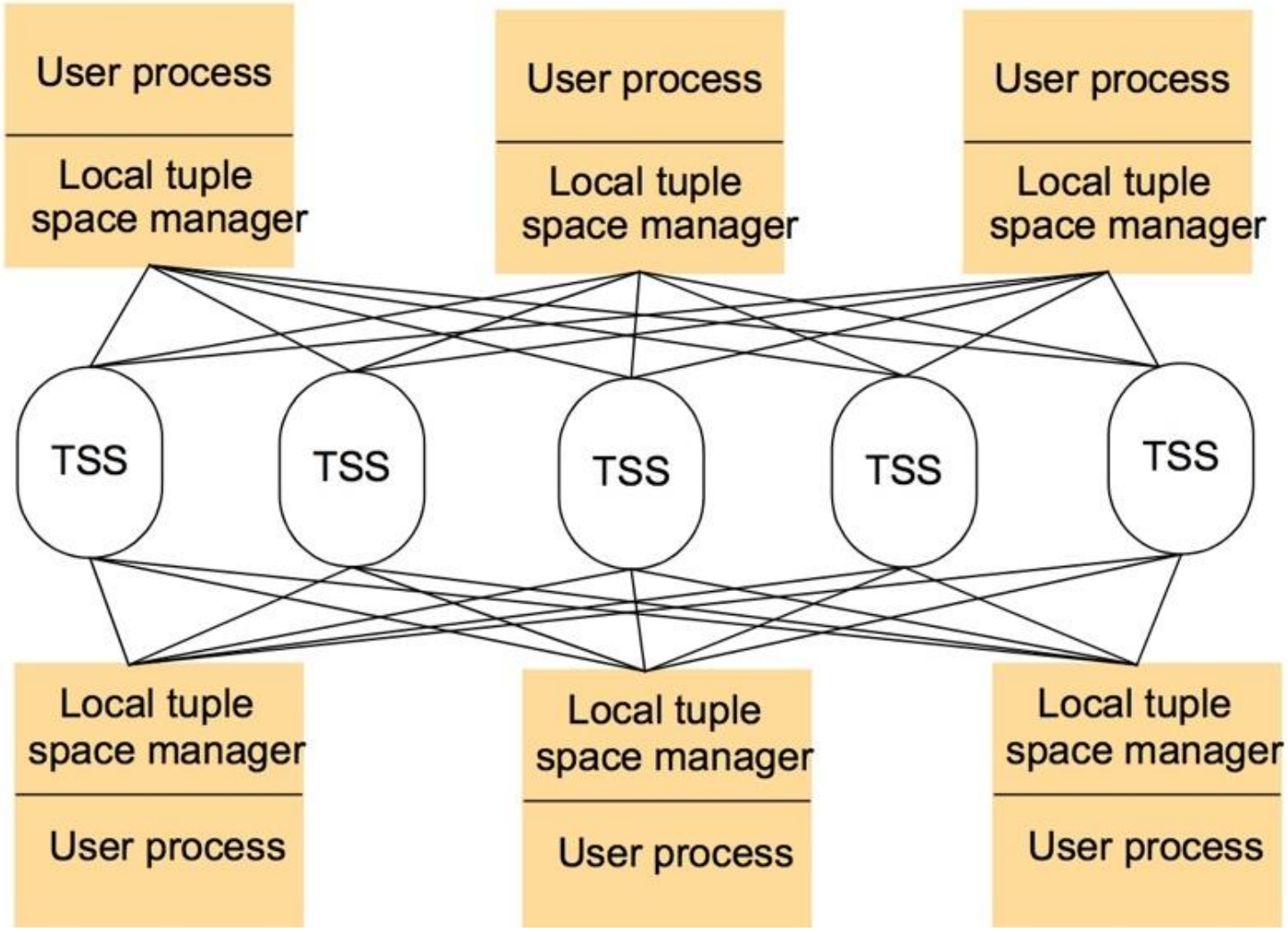
23

Properties of Tuple Spaces

- Space Uncoupling:
  - A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of a number of potential recipients.
- Time Uncoupling:
  - A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely), and hence the sender and receiver do not need to overlap in time.

24

Case Study: York Linda Kernel



Comparison of Indirect Communication 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