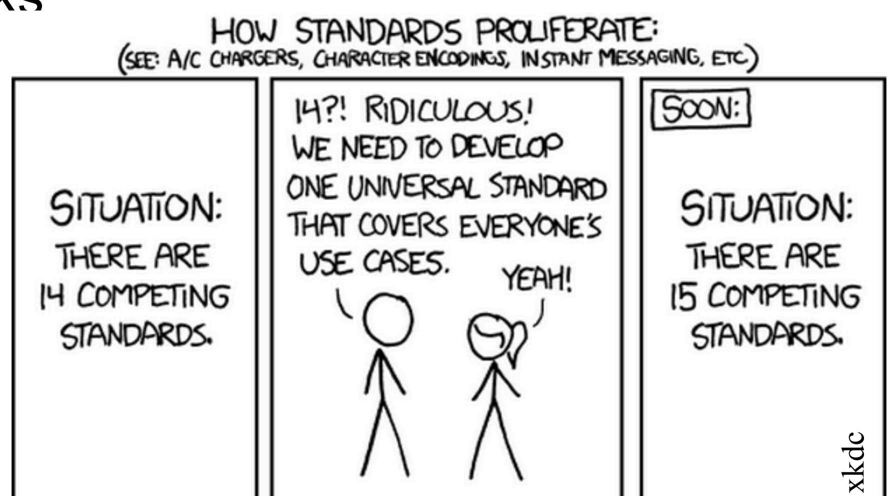
Indirect Communication

To do …

- Today
- Space and time (un)coupling
- Common techniques
- Next time: Overlay networks



Direct coupling communication

- With R-R, RPC, RMI
  - Space coupled – Sender knows the identity of the receiver and vice-versa
    - If server fails, hard to replace; clients must explicitly deal with that
  - Time coupled – Server and receiver must both exist at the time of communication

	Time-coupled
Space coupling	Communication directed to a given receiver(s) that must be available at the time *e.g. Messaging passing, RPC*

Indirect communication

- Indirect communication
  - Through an intermediary with no direct coupling between senders/receivers
  - Different types, with differences in nature of the intermediary
  - … and type of coupling

- Forms of (un)coupling
  - Space – (No) need to know the identity of the other
  - Time – (No) need to exist at the same time

Space and time (un)coupling

- Space uncoupling
  - No need to know the identity of the other party
  - Can change, update, replicate, move senders/ receivers
  - E.g., IP multicast

- Time uncoupling
  - No need to exist at the same time
  - It's ok if either party gets disconnected for a bit
  - E.g., Mailbox
  - Not the same as asynchronous – *Why not?*

Space and time un/coupling

	Time-coupled	Time-uncoupled
Space coupling	Communication directed to a given receiver(s) that must be available at the time *e.g. Messaging passing, RPC*	Sender(s) and receiver(s) can have independent lifetimes *e.g. Mailbox*
Space uncoupling	Sender does not need to know ID of receiver but they must exist at the same time *e.g. IP multicast*	Sender does not need to know ID of receiver; sender(s) and receiver(s) can have independent lifetimes *e.g. Message oriented*

Examples of indirect communication

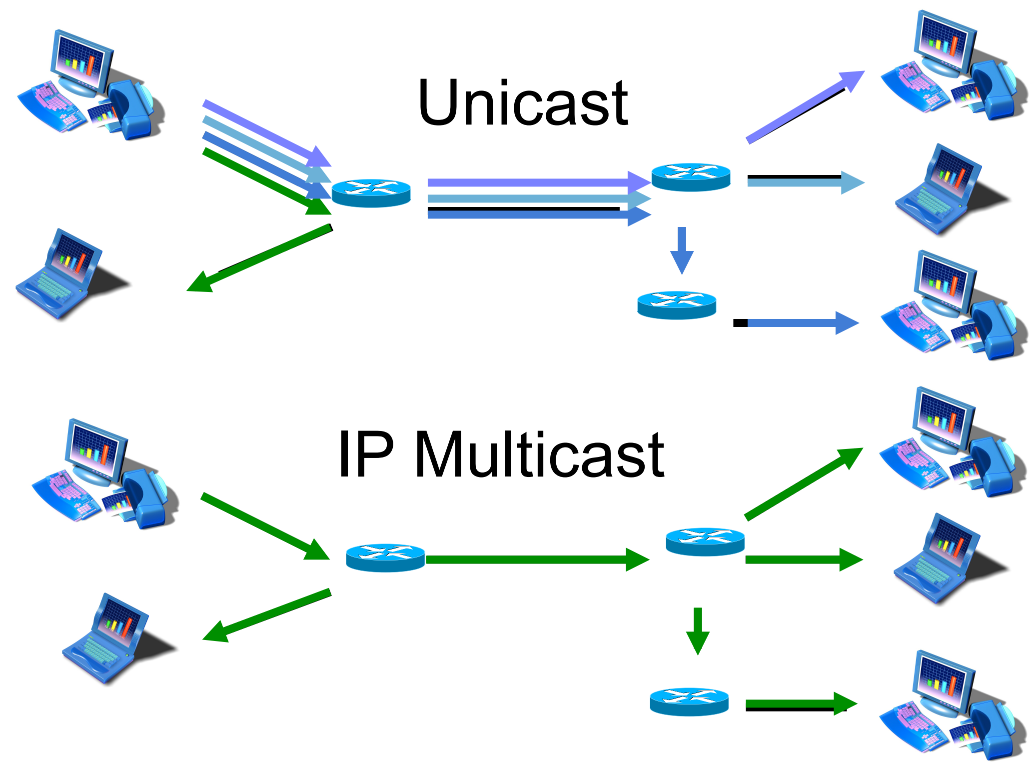
- Group communication
  - An abstraction over multicast communication
  - Space uncoupled
- Publish-subscribe systems
  - The most widely used indirect comm. techniques
  - Space uncoupled and possible time uncoupled
- Message queues
  - Space and time uncoupling through a msg queue
- Shared memory
  - Distributed shard memory and tuple spaces

Group communication

- Sender communicates with a group, as a whole, without knowing the identity of members
  – An abstraction over multicast (IP or overlay)
- Group comm. typically to process groups
  – Some work on object groups – a collection of objects, typically instances of the same class
- Some common uses
  – Reliable dissemination (e.g., financial reports)
  – Collaborative applications (e.g., multiuser games)
  – Highly-available services
  – System monitoring and management

Group communication

- Groups of processes
  - Processes can join/leave
  - A message to the group reaches all (broadcast)
- Not just programmer convenience



More efficient use of bandwidth – just once per network link

Groups and group management

- Process and object groups
  - Messages, typically unstructured byte arrays, are delivered to processes ~ socket
  - A collection of objects that process the same set of invocations concurrently; client invokes a method once on a local proxy
- Groups may be
  - Closed or open – only members or anyone can send to group
  - Overlapping or not – processes can be members of 1+ group
  - Synchronous or asynchronous
- Group membership
  - Membership service provides interface for membership changes, failure detection and notification

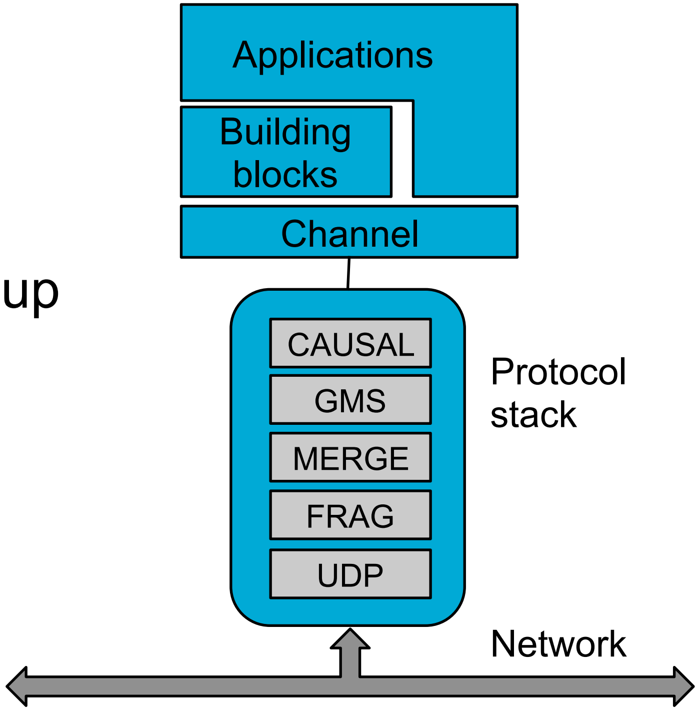
Reliable and ordered multicast

- Reliable
  - From one-to-one communication
    - Integrity – msg received is the one sent and no msg is delivered twice
    - Validity – any outgoing msg is eventually delivered
  - Agreement – if msg is delivered to one, it is delivered to all

- Ordered
  - FIFO ordering – source ordering, preserve order of the sender
  - Causal – causally related msgs arrive in the same order everywhere; if a msg *happens before* another msg this so called *causal relationship* is preserved in the delivery
  - Total ordering – All msgs arrive in the same order everywhere

JGroups toolkit as an example

- An example based on Birman and van Renesse’ work on ISIS, Horus and Ensemble
- Includes channels (handle onto a group), building blocks and a composable stack
  - Every module can be stack over/bellow any other
  - Not all stacks make sense, of course
  - CAUSAL – causal ordering
  - FRAG – configurable packetization
  - GMS – group membership system to maintain consistent view of the group
  - MERGE – Network partitions and group merges
  - …

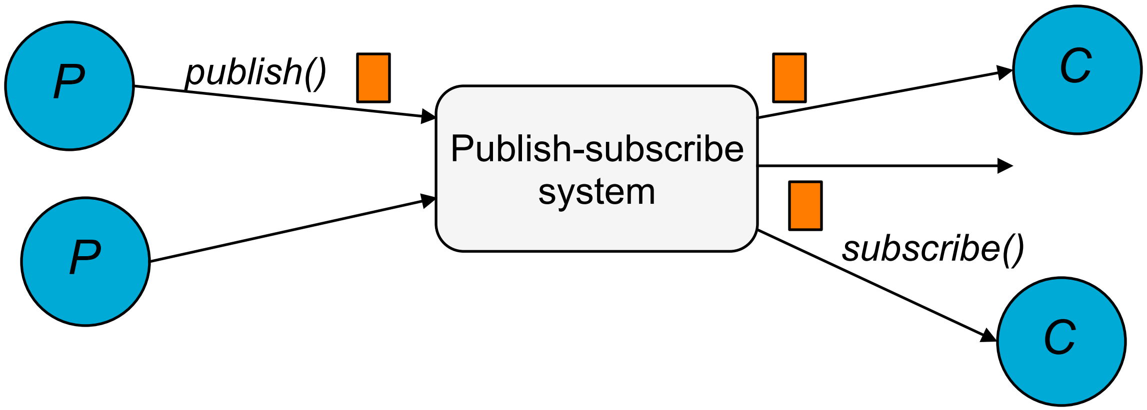


Examples of indirect communication

- Group communication
  - An abstraction over multicast communication
  - Space uncoupled
- Publish-subscribe systems
  - The most widely used indirect comm. techniques
  - Space uncoupled and possible time uncoupled
- Message queues
  - Space and time uncoupling through a msg queue
- Shared memory
  - Distributed shard memory and tuple spaces

Publish-subscribe

- AKA distributed event-based systems
  - The most widely used of all indirect communication models
  - E.g., CORBA Events, TIB Rendezvous, Scribe, Echo, …
- Publishers and subscribers
  - Publishers publish events, subscribers subscribe to them
- The pub/sub system job
  - Match subscriptions with published events, ensure delivery



- Example applications
  - financial systems, live feeds, monitoring apps, …

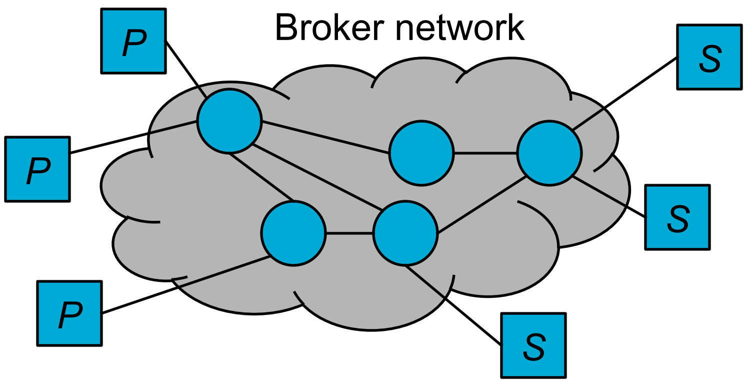
Programming and subscription models

- Simple programming model
  - Publishers *publish(e)*, subscribes *subscribe(f)* where *f* is a filter on the type of events they care for, *unsubscribe()*

- Different subscription models
  - Channel-based
    - Basic, publishing to named channels
  - Topic or subject based
    - Notifications are expressed in terms of a number of fields; one field denotes the topic
  - Content-based
    - Allows subscription over a range of fields
  - Other types explored
    - Type-, context- and concept-based and more complex event processing

Publish-subscribe – implementation

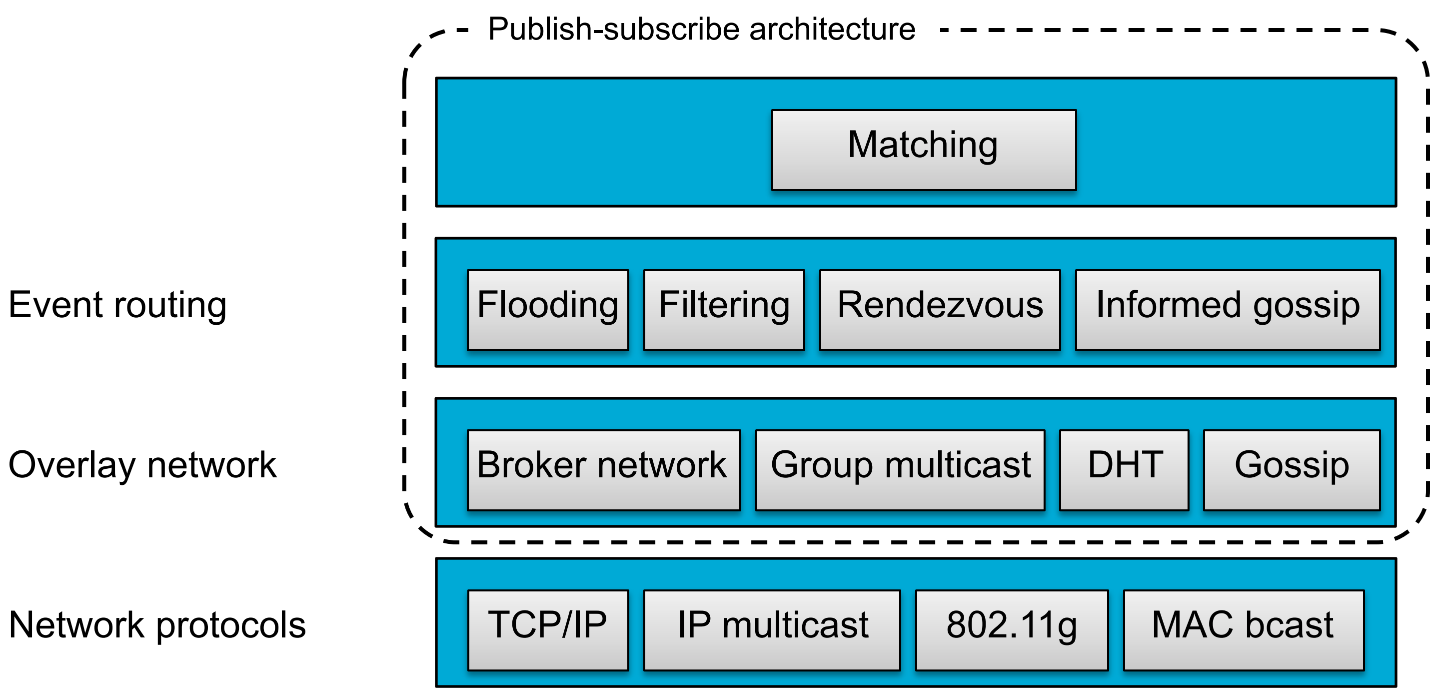
- Goal – efficient delivery of the right events to the right subscribers with appropriate security considerations
- Some design options
  - Centralized/distributed
    - Centralized event broker
    - Network of brokers



    - Full P2P – not distinction between publishers and subscribers, i.e., everyone is a broker
  - Routing options …

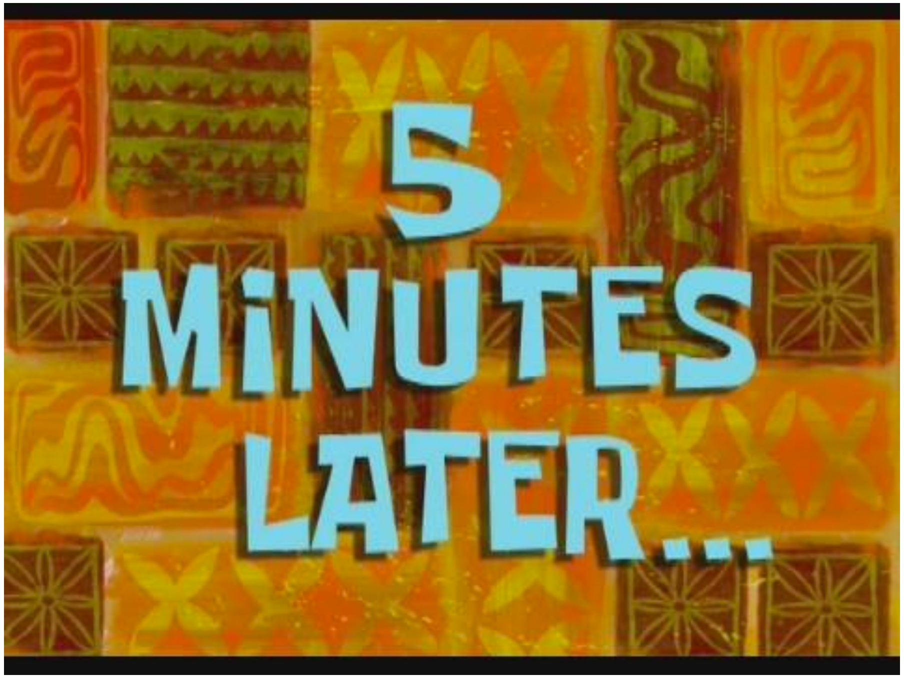
Architecture of publish-subscribe systems

- Routing options
  - Flooding – send to all, matching done at the subscriber
  - Filtering – every node in the network of brokers does filtering-based routing
  - Rendezvous – a node responsible for matching notifications and subscribers



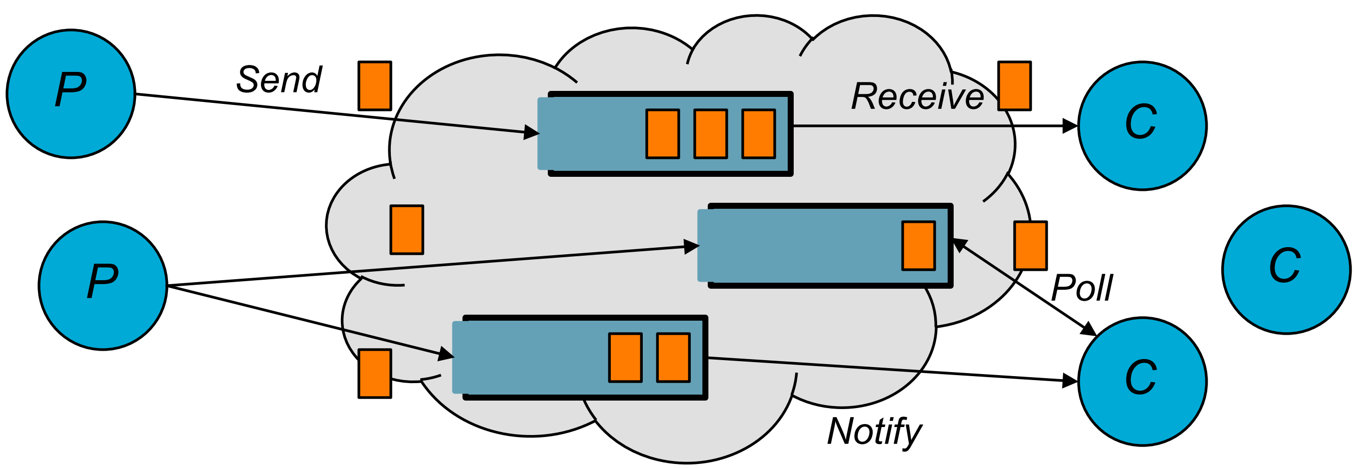
Back in 5'

- … message queues and shared memory



(Distributed) message queues

- Point-to-point comm. through an intermediary queue
  - Senders place msgs into a queue, receivers removed them
    - Queues correspond to buffers at communication servers
  - E.g., IBM WebSphere MQ, Java Messaging Service, Oracle's Stream Advanced Queuing



(Distributed) message queues

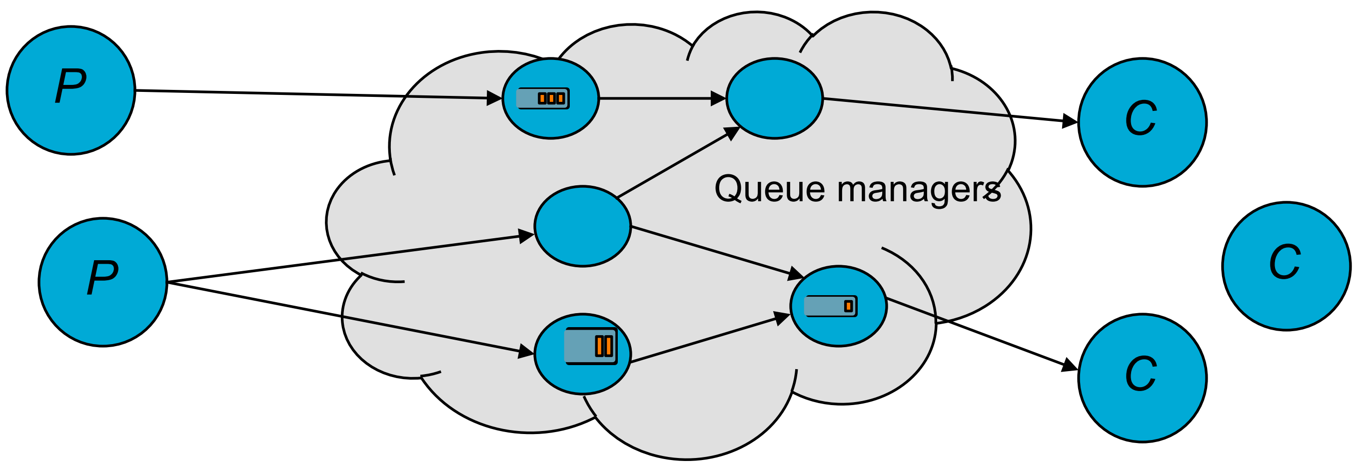
- Details on messages
  - Typically include dest queue, priority, delivery mode, and body
    - In Oracle's AQ, messages are rows in a DB table/queue
  - Messages are persistent
  - Typical queuing policies FIFO and priority-based

- Use for app integration – broker takes care of application heterogeneity
  - Transforms incoming messages to target format
  - Often acts as an application gateway
  - May provide subject-based routing capabilities

(Distributed) message queues

- Styles of receive
  - Blocking – Block until an appropriate message is available
  - Non-blocking – Polling to see if a message is available
  - Notify – Notify when a message arrives

- Centralized and distributed message queues
  - In WebSphere MQ, queues are managed by *queue managers*
  - Queue managers can be inter-connected as brokers in pub-sub



Examples of indirect communication

- Group communication
  - An abstraction over multicast communication
  - Space uncoupled
- Publish-subscribe systems
  - The most widely used indirect comm. techniques
  - Space uncoupled and possible time uncoupled
- Message queues
  - Space and time uncoupling through a msg queue
- Shared memory
  - Distributed shard memory and tuple spaces

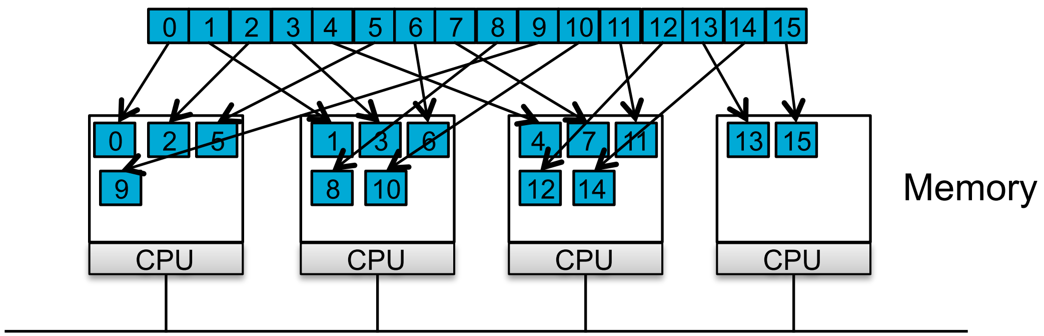
Shared memory approaches

- Distributed share memory
  - Allow networked computers to share memory
  - How to make distributed memory appear local?
  - Leverage MMU
    - Page fault handler invokes DSM protocol
    - Bring page from a remote node instead of from HD
  - Of course, underneath it all – message passing

- Compare with message passing
  - No need to marshalled/unmarshalled data, send/receive, …
  - Synchronization via typical shared-memory programming constructs like locks
  - Potentially easier to program for with a performance cost

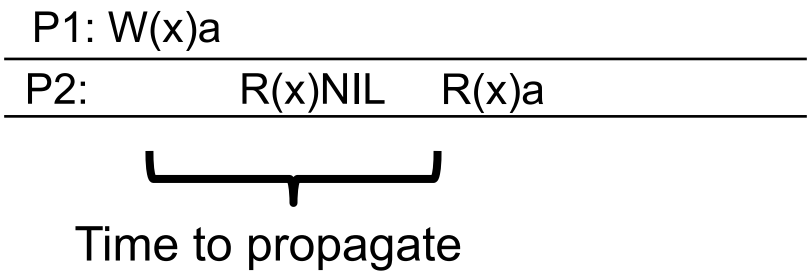
Shared memory approaches

- Simplest design
  - Each virtual page in one machine at a time (no caching)
  - A directory keeps track of things, potentially a bottleneck
  - Distributed directory – hash(page#)
  - Design issues
    - Size of the page
    - Caching and consistency models



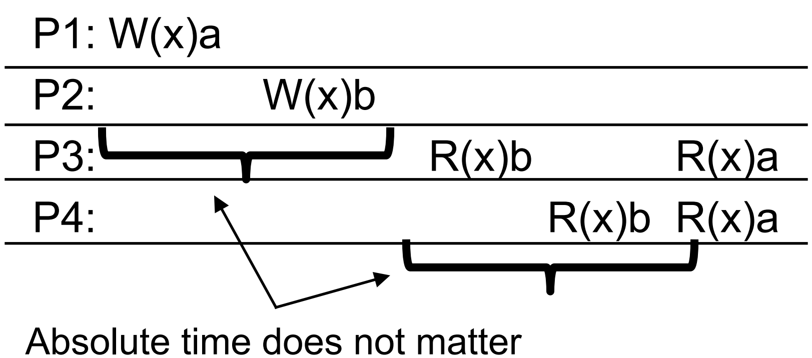
Shared memory and consistency

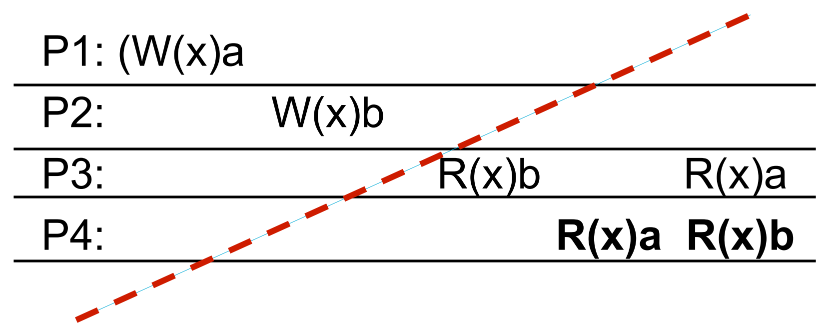
- Consistency model
  - When modifications to data may be seen at a given processor
  - Defines the programmer's view, placing restrictions on what values can be returned by a read (a contract)
  - Determines what optimizations are possible
- E.g., sequential consistency
  - Some basic notation
    - $W_i(x)a$ – process P_i wrote value a to x
    - $R_i(x)b$ – process P_i read value b from x



Sequential consistency

- *Result of execution – as if*
  - *operations of all processes were executed in some sequential order, and …*
  - *the operations of each process appear in this sequence in the order specified by its program*
  - i.e., Any valid interleaving of ops is OK, but all processes see the same interleaving





- Lineralizable – interleaving is consistent with real time at which operations occurred in the actual execution

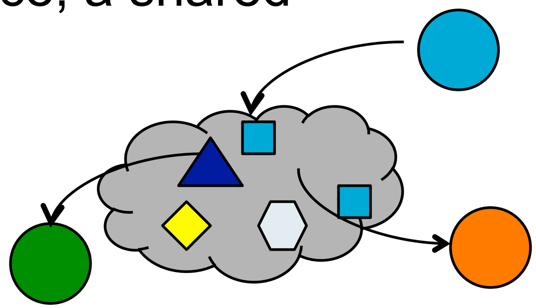
The burden of sequential consistency

- Processor must ensure that previous memory operation is complete before proceeding to the next
- So …
  - Determine completion of write; get ack for all
  - If caching, write invalidates or updates all cached copies
  - Hold off on read requests until all writes are complete
- Maybe we can relax this a bit if next steps don't depend on the value
  - Causal consistency …

Shared memory – tuple spaces

- First introduced with *Linda* by D. Gelernter
  - Adopted by IBM Tspaces, JavaSpaces, etc.

- Programming model
  - Processes communicate through a tuple space, a shared collection of tuples
  - Tuple – a sequence of 1+ typed data fields
  - Operations
    - Write – adds a tuple
    - Read – returns the value of a tuple w/o changing the tuple space
    - Take – returns the value of a tuple and removes the tuple
    - For read/take, give a template; system returns a tuple that matches
  - Tuples are immutable - to modify a tuple, take it and write a new one



Shared memory – tuple spaces

- Original Linda model had a single, global tuple space
  - Not optimal for a large system, e.g., aliasing of tuples
  - Aliasing – read/take matching tuples by accident
  - Following systems use multiple tuple spaces and some allow the dynamic creation of tuple spaces

- Linda was anticipated as a centralized system
  - Performance and reliability concerns
  - Following systems support distributed tuple spaces
  - Different approaches from state machine replication and tuple-specific approaches to simple partitioning of the tuple space

Summary

- The power of indirection in communication – communication through an intermediary
  - Uncoupling in space and/or time

	Group	Pub/sub	MQ	DSM	Tuples
Space uncoupled	Yes	Yes	Yes	Yes	Yes
Time uncoupled	Possible	Possible	Yes	Yes	Yes
Style	Comm	Comm	Comm	State	State
Comm pattern	1-m	1-m	1-1	1-m	1-1/1-m
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based only	No	No	Yes