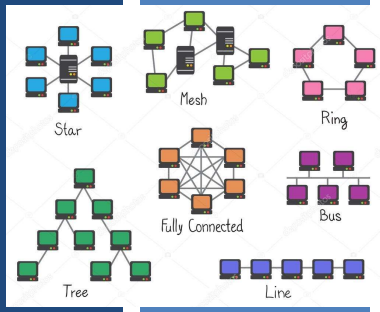
TCSS 558: APPLIED DISTRIBUTED COMPUTING



Chapter 4 – Communication
Chapter 6 - Coordination

Wes J. Lloyd
School of Engineering
and Technology
University of Washington - Tacoma

OBJECTIVES

- Homework 2
- Chapter 4 Communication
  - 4.3 Message-oriented communication
  - 4.4 Multicast communication
- Ch. 6 – Coordination
  - 6.1 Clock synchronization
  - 6.2 Logical clocks, Lamport clocks, Vector clocks
  - 6.3 Distributed mutual exclusion

FEEDBACK – 2/27

- **What is the difference between "precopy" and "on demand" VM migration?**
- **PRECOPY** – before VM migration, memory pages are copied to the destination host on demand
  - System must track pages modified on original VM than must be updated
- **ON DEMAND** – VM is immediately migrated, memory pages are only copied to the destination host when they are accessed
  - Programs access pages by their memory address
  - System goes to fetch them, but they are blanks
  - Blank fetch triggers retrieval from remote machine
  - Requires keeping original VM around for a long time

FEEDBACK - 2

- **Could you please explain more on the three types of blocking?**
- For Synchronous communication client blocks and waits
- For each level, client increasingly blocks for longer
- **Three types of blocking**
  1. Until middleware notifies it will take over delivering request
     client → proxy-server → server
     client blocks until proxy routes request
  2. Sender may synchronize until request has been delivered (**for long request, large data payload**)
     client → [BIG DATA] → server
  3. Sender waits until request is processed and result is returned (**full**)
     client ←→ server (fully synchronous)

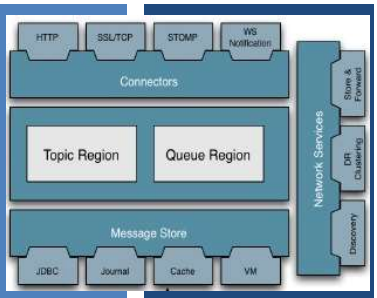
FEEDBACK - 3

- **How is total latency calculated before deciding on an efficient route?**
- Routing across internet is between regions not nodes
- Traffic first routed to region, then to specific nodes (IPs)
- Known as hierarchical routing
- Routing adaption occurs at different timescales

Mechanism	Timescale	Adapt/Respond to
Load-sensitive routing	Seconds	Traffic hotspots
Routing	Minutes	Equipment failures
Traffic engineering	Hours	Network load (e.g. Netflix…)
HW Provisioning	Months	Network customers

Based on https://courses.cs.washington.edu/courses/cse461/17au/lectures/06-1-routing.pdf

CH. 4.3: MESSAGE-ORIENTED COMMUNICATION



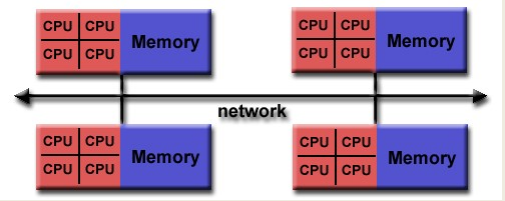
Apache ActiveMQ

4.3 - MESSAGE ORIENTED COMMUNICATION

- **Topics**
- Message passing interface (MPI)
- Message oriented middleware
- Message queueing systems
- Advanced message queueing protocol (AMQP)

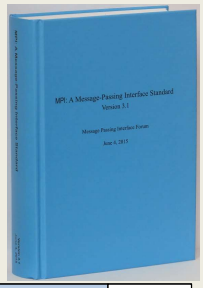
MESSAGE PASSING INTERFACE (MPI)

- MPI introduced – version 1.0 March 1994
- Message passing API for parallel programming: ***supercomputers***
- Communication protocol for parallel programming for: Supercomputers, High Performance Computing (HPC) clusters
- Point-to-point and collective communication
- Goals: high performance, scalability, portability
- Most implementations in C, C++, Fortran
- OpenMPI – open source x86 implementation



MOTIVATIONS FOR MPI

- Motivation: sockets insufficient for interprocess communication on large scale HPC compute clusters and super computers
  - Sockets at the wrong level of abstraction
  - Sockets designed to communicate over the network using general purpose TCP/IP stacks
  - Not designed for proprietary protocols
  - Not designed for high-speed interconnection networks used by supercomputers, HPC-clusters, etc.
  - Better buffering and synchronization needed



MOTIVATIONS FOR MPI - 2

- Supercomputers had proprietary communication libraries
  - Offer a wealth of efficient communication operations
- All libraries mutually incompatible
- Led to significant portability problems developing parallel code that could migrate across supercomputers
- Led to development of MPI
  - To support transient (non-persistent) communication for parallel programming

MPI FUNCTIONS / DATATYPES

- Very large library, v1.0 (1994) 128 functions
- Version 3 (2015) 440+
- MPI data types:
- Provide common mappings

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI_ABORT	MPI_ADDRESS	MPI_ALLGATHER	MPI_ALLGATHERV
MPI_ALLREDUCE	MPI_ALLTOALL	MPI_ALLTOALLV	MPI_ATTR_DELETE
MPI_ATTR_GET	MPI_ATTR_PUT	MPI_BARRIER	MPI_BCAST
MPI_BSEND	MPI_BSEND_INIT	MPI_BUFFER_ATTACH	MPI_BUFFER_DETACH
MPI_CANCEL	MPI_CARTDIM_GET	MPI_CART_COORDS	MPI_CART_CREATE
MPI_CART_GET	MPI_CART_MAP	MPI_CART_RANK	MPI_CART_SHIFT
MPI_CART_SUB	MPI_COMM_COMPARE	MPI_COMM_CREATE	MPI_COMM_DUP
MPI_COMM_FREE	MPI_COMM_GROUP	MPI_COMM_RANK	MPI_COMM_REMOTE_GROUP
MPI_COMM_REMOTE_SIZE	MPI_COMM_SIZE	MPI_COMM_SPLIT	MPI_COMM_TEST_INTER
MPI_DIMS_CREATE	MPI_ERRHANDLER_CREATE	MPI_ERRHANDLER_FREE	MPI_ERRHANDLER_GET
MPI_ERRHANDLER_SET	MPI_ERROR_CLASS	MPI_ERROR_STRING	MPI_FINALIZE
MPI_GATHER	MPI_GATHERV	MPI_GET_COUNT	MPI_GET_ELEMENTS
MPI_GET_PROCESSOR_NAME	MPI_GRAPHDIMS_GET	MPI_GRAPH_CREATE	MPI_GRAPH_GET
MPI_GRAPH_MAP	MPI_GRAPH_NEIGHBORS	MPI_GRAPH_NEIGHBORS_COUNT	MPI_GROUP_COMPARE
MPI_GROUP_DIFFERENCE	MPI_GROUP_EXCL	MPI_GROUP_FREE	MPI_GROUP_INCL
MPI_GROUP_INTERSECTION	MPI_GROUP_RANGE_EXCL	MPI_GROUP_RANGE_INCL	MPI_GROUP_RANK
MPI_GROUP_SIZE	MPI_GROUP_TRANSLATE_RANKS	MPI_GROUP_UNION	MPI_IBSEND
MPI_INIT	MPI_INITIALIZED	MPI_INTERCOMM_CREATE	MPI_INTERCOMM_MERGE
MPI_IPROBE	MPI_IRECV	MPI_IRSEND	MPI_ISEND
MPI_ISSEND	MPI_KEYVAL_CREATE	MPI_KEYVAL_FREE	MPI_OP_CREATE
MPI_OP_FREE	MPI_PACK	MPI_PACK_SIZE	MPI_PCONTROL
MPI_PROBE	MPI_RECV	MPI_RECV_INIT	MPI_REDUCE
MPI_REDUCE_SCATTER	MPI_REQUEST_FREE	MPI_RSEND	MPI_RSEND_INIT
MPI_SCAN	MPI_SCATTER	MPI_SCATTERV	MPI_SEND
MPI_SENDRECV	MPI_SENDRECV_REPLACE	MPI_SEND_INIT	MPI_SSEND
MPI_SSEND_INIT	MPI_START	MPI_STARTALL	MPI_TEST
MPI_TESTALL	MPI_TESTANY	MPI_TESTSOME	MPI_TEST_CANCELLED
MPI_TOPO_TEST	MPI_TYPE_COMMIT	MPI_TYPE_CONTIGUOUS	MPI_TYPE_EXTENT
MPI_TYPE_FREE	MPI_TYPE_HINDEXED	MPI_TYPE_HVECTOR	MPI_TYPE_INDEXED
MPI_TYPE_LB	MPI_TYPE_SIZE	MPI_TYPE_STRUCT	MPI_TYPE_UB
MPI_TYPE_VECTOR	MPI_UNPACK	MPI_WAIT	MPI_WAITALL
MPI_WAITANY	MPI_WAITSOME	MPI_WTICK	MPI_WTIME

COMMON MPI FUNCTIONS

- MPI - no recovery for process crashes, network partitions
- Communication among grouped processes: `(groupID, processID)`
- IDs used to route messages in place of IP addresses

Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send message, wait until copied to local/remote buffer
MPI_ssend	Send message, wat until transmission starts
MPI_sendrecv	Send message, wait for reply
MPI_isend	Pass reference to outgoing message and continue
MPI_issend	Pass reference to outgoing messages, wait until receipt start
MPI_recv	Receive a message, block if there is none
MPI_irecv	Check for incoming message, **do not block!**

MESSAGE-ORIENTED-MIDDLEWARE

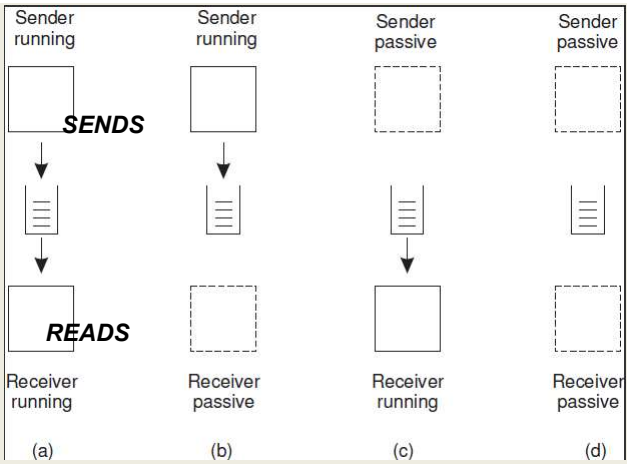
- **Message-queueing systems**
  - Provide extensive support for ***persistent*** asynchronous communication
  - In contrast to transient systems
  - Temporally decoupled: messages are eventually delivered to recipient queues
- Message transfers may take minutes vs. sec or ms
- Each application has its own private queue to which other applications can send messages

MESSAGE QUEUEING SYSTEMS: USE CASES

- Enables communication between applications, or sets of processes
  - User applications
  - App-to-database
  - To support distributed real-time computations
- Use cases
  - Batch processing, Email, workflow, groupware, routing subqueries

MESSAGE QUEUEING SYSTEMS

- Scenarios:
  (a) Sender/receiver both running
  (b) Sender running, receiver offline
  (c) Sender offline, receiver running
  (d) Sender/receiver both offline
- Queue persists msgs, and attempts to send them but no one may be available to receive them…



MESSAGE QUEUEING SYSTEMS - 2

- **Key:** Truly persistent messaging
- Message queueing systems can persist messages for awhile and senders and receivers can be offline
- **Messages**
- Contain *any* data, may have size limit
- Are properly addressed, to a destination queue
- **Basic Inteface**
- PUT: called by sender to append msg to specified queue
- GET: blocking call to remove oldest msg from specified queue
  - Blocked if queue is empty
- POLL: Non-blocking, gets msg from specified queue

MESSAGE QUEUEING SYSTEMS ARCHITECTURE

- **Basic interface cont'd**
- NOTIFY: install a callback function, for when msg is placed into a queue. Notifies receivers

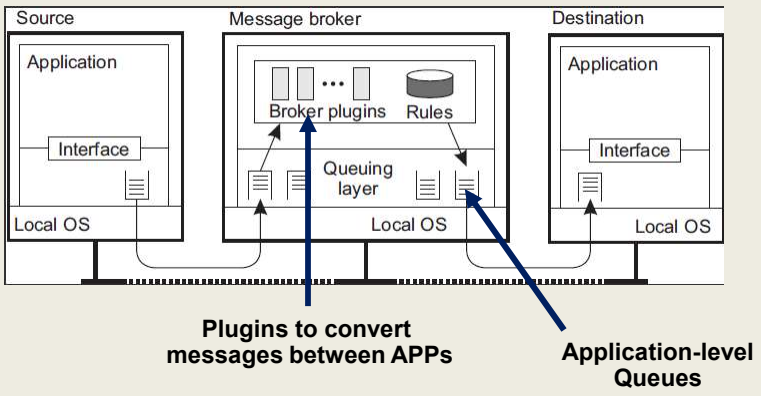
- **Queue managers**: manage individual message queues as a separate process/library
- Applications get/put messages only from ***local*** queues
- Queue manager and apps share local network
- **ISSUES:**
- How should we reference the destination queue?
- How should names be resolved (looked-up)?
  - Contact address (host, port) pairs
  - Local look-up tables can be stored at each queue manager

MESSAGE QUEUEING SYSTEMS ARCHITECTURE - 2

- **ISSUES**:
- How do we route traffic between queue managers?
  - How are name-to-address mappings efficiently kept?
  - Each queue manager should be known to all others

- **Message brokers**
- Handle message conversion among different users/formats
- Addresses cases when senders and receivers don't speak the same protocol (language)
- Need arises for message protocol converters
  - "Reformatter" of messages
- Act as application-level gateway

MESSAGE BROKER ORGANIZATION



AMQP PROTOCOL

- Message-queueing systems initially developed to enable legacy applications to interoperate
- Decouple inter-application communication to "open" messaging-middleware
- Many are proprietary solutions, ***so not very open***
- e.g. Microsoft Message Queueing service, Windows NT 1997
- **Advanced message queueing protocol (AMQP)**, 2006
- Address openness/interoperability of proprietary solutions
- Open wire protocol for messaging with powerful routing capabilities
- Help *abstract* messaging and application interoperability by means of a generic open protocol
- Suffer from incompatibility among protocol versions
- pre-1.0, 1.0+

AMQP - 2

- Consists of: Applications, Queue managers, Queues
- **Connections:** set up to a queue manager, TCP, with potentially many channels, stable, reused by many channels, long-lived
- **Channels:** support short-lived one-way communication
- **Sessions:** bi-directional communication across two channels
- **Link:** provide fine-grained flow-control of message transfer/status between applications and queue manager

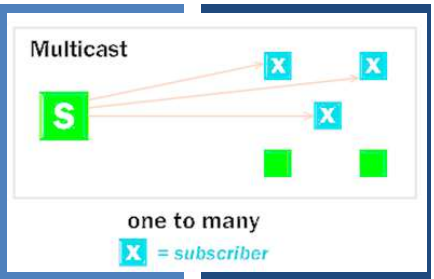
AMQP MESSAGING

- AMQP nodes: producer, consumer, queue
- Producer/consumer: represent regular applications
- Queues: store/forward messages

- Persistent messaging:
- **Messages** can be marked ***durable***
- These messages can only be delivered by nodes able to recover in case of failure
- Non-failure resistant nodes must reject durable messages
- **Source/target** nodes can be marked ***durable***
- Track what is durable (node state, node+msgs)

MESSAGE-ORIENTED-MIDDLEWARE EXAMPLES:

- **Some examples:**
- RabbitMQ, Apache QPid
  - Implement Advanced Message Queueing Protocol (AMQP)
- Apache Kafka
  - **Dumb broker** (message store), similar to a distributed log file
  - **Smart consumers** – intelligence pushed off to the clients
  - Stores stream of records in categories called topics
  - Supports voluminous data, many consumers, with minimal O/H
  - Kafka **does not track** which messages were read by each consumer
  - Messages are removed after timeout
  - Clients must track their own consumption (*Kafka doesn't help*)
  - Messages have key, value, timestamp
  - Supports high volume pub/sub messaging and streams



CH. 4.4: MULTICAST COMMUNICATION

Apache ActiveMQ

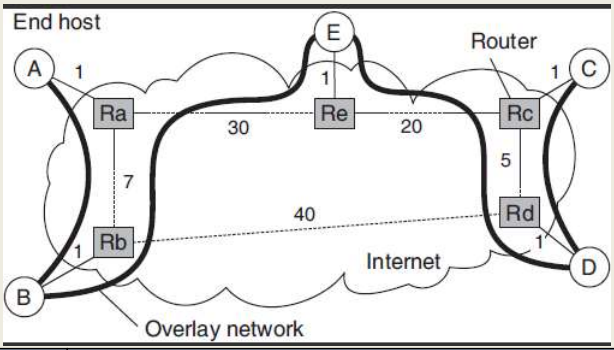
MULTICAST COMMUNICATION

- Sending data to multiple receivers
- Many ***failed*** proposals for network-level / transport-level protocols to support multicast communication
- **Problem:** How to set up communication paths for information dissemination?
- **Solutions:** require huge management effort, human invention

- Focus shifted more recently to **peer-to-peer** networks
  - Structured overlay networks can be setup easily and provide efficient communication paths
  - Application-level multicasting techniques more successful
  - Gossip-based dissemination: unstructured p2p networks

NETWORK STRUCTURE

- **Overlay network**
  - Virtual network implemented on top of an actual physical network
- **Underlying network**
  - The actual physical network that implements the overlay



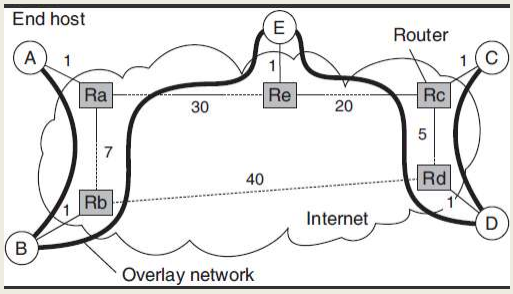
APPLICATION LEVEL TREE-BASED MULTICASTING

- Application level multi-casting
  - Nodes organize into an overlay network
  - Network routers not involved in group membership
  - Group membership is managed at the application level (A2)
- Downside:
  - Application-level routing likely less efficient than network-level
  - Necessary tradeoff until having better multicasting protocols at lower layers
- Overlay topologies
  - TREE: top-down, unique paths between nodes
  - MESH: nodes have multiple neighbors; multiple paths between nodes

MULTICAST TREE METRICS

- Measure quality of application-level multicast tree
- **Link stress:** is defined per link, counts how often a packet crosses same link (***ideally not more than 1***)
- **Stretch**: ratio in delay between two nodes in the **overlay** vs. the **underlying** networks

Numbers represent network delay between nodes



MULTICAST TREE METRICS - 2

- **Stretch (Relative Delay Penalty RDP)**
- CONSIDER routing from B to C
- ***What is the Stretch?***
- Stretch (delay ratio) = Overlay-delay / Underlying-delay
- *Overlay:* B→Rb→Ra→Re→E→Re→Rc→Rd→D→Rd→Rc→ C = 73
- *Underlying:* B→Rb→Rd→Rc→C = 47
- Stretch = 73 / 47 = 1.55

- **Tree cost:** Overall cost of the overlay network
- Ideally would like to minimize network costs
- Find a minimal spanning tree which minimizes total time for disseminating information

FLOOD-BASED MULTICASTING

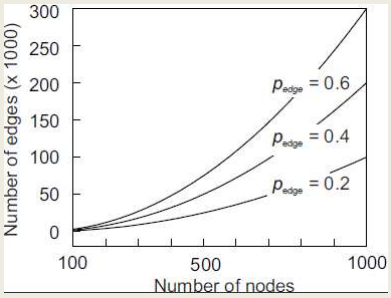
- Broadcasting: every node in overlay receives message

- Key design issue: minimize the use of intermediate nodes for which the message is not intended
- Tree: if only the leaf nodes are to receive the multicast message, many intermediate nodes are involved
- Solution: construct an overlay network for each multicast group

- **Flooding**: each node simply forwards a message to each of its neighbors, except to the message originator

RANDOM GRAPHS

- When no information on the structure of the overlay network
- Assume network can be represented as a **Random graph**
- Probability P_{edge} that two nodes are joined
- Overlay network will have: ½ * P_{edge} * N * (N-1) edges



PROBABILISTIC FLOODING

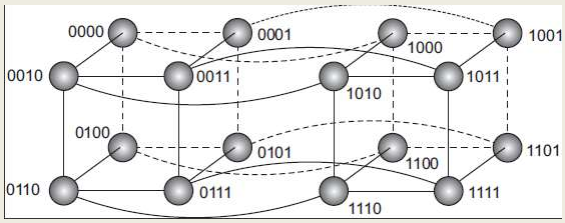
- ***….Washington state in winter?***

- When a node is flooding a message, concept is to enforce a probability of message spread (p_{flood})
- Throttles message flooding based on a probability
- Implementation needs to consider # of neighbors to achieve various p_{flood} scores
- With lower p_{flood} messages may not reach all nodes

- **USEFULNESS:** For random network with 10,000 nodes
- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

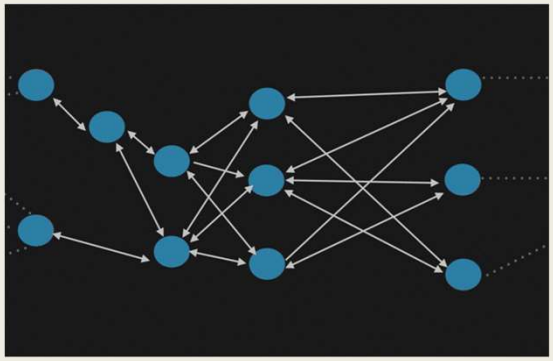
MESSAGE FLOODING

- For deterministic topologies (such as hypercube), design of efficient flooding scheme is much simpler
- If the overlay network is structured, this gives us a deterministic topology
- **Hypercube:** nodes forward only to higher dimension nodes
- N(1001) broadcast will only go to N(1011) and N(1000)
- Broadcast requires just: N-1 messages, where nodes $N=2^n$, n=dimensions of hypercube



GOSSIP BASED DATA DISSEMINATION

- When structured peer-to-peer topologies are not available
- Gossip based approaches support multicast communication over unstructured peer-to-peer networks
- General approach is to leverage how gossip spreads across a group
- This is also called "epidemic behavior"…
- Data updates for a specific item begin at a specific node



INFORMATION DISSEMINATION

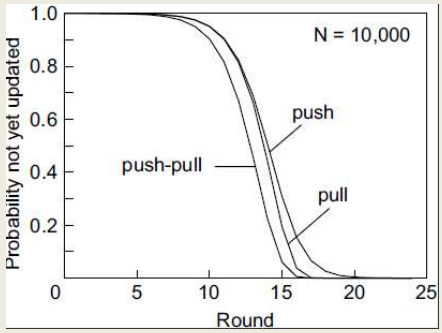
- **Epidemic algorithms**: algorithms for large-scale distributed systems that spread information
- Goal: "infect" all nodes with new information as fast as possible
- **Infected**: node with data that can spread to other nodes
- **Susceptible**: node without data
- **Removed**: node with data that is unable to spread data

ANTI ENTROPY DISSEMINATION MODEL

- **Anti-entropy:** Propagation model where node P picks node Q at random and exchanges message updates
- Akin to random walk
- **PUSH:** P only **pushes** its own updates to Q
- **PULL:** P only **pulls** in new updates from Q
- **TWO-WAY:** P and Q send updates to each other (i.e. a push-pull approach)
- Push only: hard to propagate updates to last few hidden susceptible nodes
- Pull: better because susceptible nodes can pull updates from infected nodes
- Push-pull is better still

ANTI ENTROPY EFFECTIVENESS

- **Round**: span of time during which every node takes initiative to exchange updates with a randomly chosen node
- The number of rounds to propagate a single update to all nodes requires O(log(N)), where N=number of nodes
- Let p_i denote probability that node P has not received msg m after the i^{th} round.
- For pull, push, and push-pull based approaches:



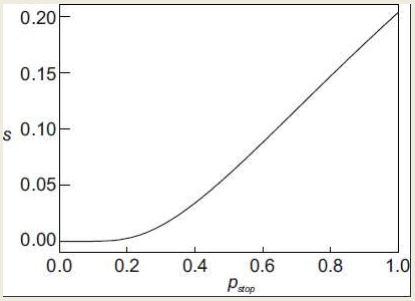
RUMOR SPREADING

- Variant of epidemic protocols
- Provides an approach to "stop" message spreading
- Mimics "gossiping" in real life

- **Rumor spreading:**
- **Node P** receives new data **item X**
- Contacts an arbitrary **node Q** to push update
- **Node Q** reports already receiving **item X** from another node
- **Node P** may loose interest in spreading the rumor with probability = p_{stop}, let's say 20% . . . (or 0.20)

RUMOR SPREADING - 2

- Does not guarantee all nodes will be updated
- The fraction of nodes s, that remain susceptible grows relative to the probability that node P stops propagating when finding a node already having the message
- Fraction of nodes not updated remains < 0.20 with high p_{stop}
- Susceptible nodes (s) vs. probability of stopping →



DIRECTIONAL GOSSIPING

- Taking network topology into account can help
- When gossiping, nodes connected to only a few other nodes are more likely to be contacted
- **Epidemic protocols assume:**
- For gossiping nodes are randomly selected
- One node, can randomly select any other node in the network
- Complete set of nodes is known to each member

REMOVING DATA

- Gossiping is good for spreading data
- **But how can data be removed from the system?**
- Idea is to issue ***"death certificates"***
- Act like data records, which are spread like data
- When death certificate is received, data is deleted
- Certificate is held to prevent data element from reinitializing from gossip from other nodes
- Death certificates time-out after expected time required for data element to clear out of entire system
- A few nodes maintain death certificates forever

DEATH CERTIFICATE EXAMPLE

- **For example:**
- **Node P** keeps death certificates forever
- **Item X** is removed from the system
- **Node P** receives an update request for **Item X**, but ***also*** holds the death certificate for **Item X**
- **Node P** will recirculate the death certificate across the network for **Item X**

CHAPTER 6 - COORDINATION

- 6.1 Clock Synchronization
  - Physical clocks
  - Clock synchronization algorithms
- 6.2 Logical clocks
  - Lamport clocks
  - Vector clocks
- 6.3 Mutual exclusion
- 6.4 Election algorithms
- 6.6 Distributed event matching *(light)*
- 6.7 Gossip-based coordination *(light)*

CHAPTER 6 - COORDINATION

- How can processes synchronize and coordinate data?
- Process synchronization
  - Coordinate cooperation to grant individual processes temporary access to shared resources (e.g. a file)
- Data synchronization
  - Ensure two sets of data are the same (data replication)
- Coordination
  - Goal is to manage interactions and dependencies between activities in the distributed system
  - Encapsulates synchronization

COORDINATION - 2

- Synchronization challenges begin with **time**:
  - How can we synchronize computers, so they all agree on the time?
  - How do we measure and coordinate when things happen?
- Fortunately, for synchronization in distributed systems, it is often sufficient to only agree on a relative ordering of events
  - E.g. not actual time

COORDINATION - 3

- Groups of processes often appoint a **coordinator**
- **Election algorithms** can help elect a leader
- Synchronizing access to a shared resource is achieved with **distributed mutual exclusion** algorithms
- Also in chapter 6:
  - Matching subscriptions to publications in publish-subscribe systems
  - Gossip-based coordinate problems:
    - Aggregation
    - Peer sampling
    - Overlay construction

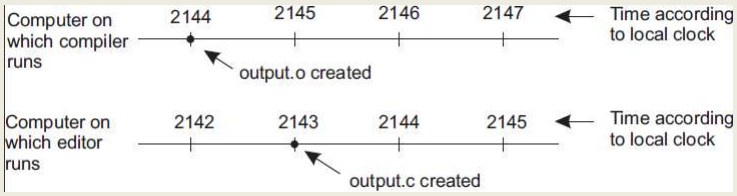
CH. 6.1: CLOCK SYNCHRONIZATION

L13.47

CLOCK SYNCHORNIZATION

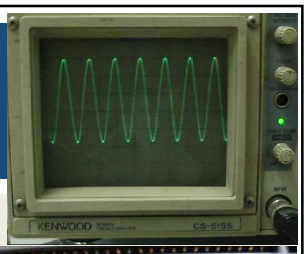
- Example:
- "make" is used to compile source files into binary object and executable files
- As an optimization, make only compiles files when the "last modified time" of source files is more recent that object and executables
- Consider if files are on a shared disk of a distributed system where there is no agreement on time
- Consider if the program has 1,000 source files

TIME SYNCHRONIZATION PROBLEM FOR DISTRIBUTED SYSTEMS



- Updates from different machines, may have clocks set to different times
- Make becomes confused with which files to recompile

PHYSICAL CLOCKS

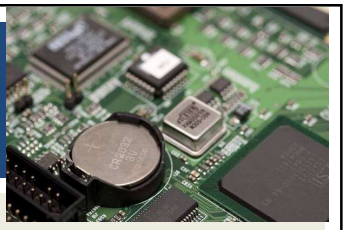


- **Computer timers:** precisely machined quartz crystals
- When under tension, they oscillate at a well defined frequency
- In analog electronics/communications crystals once used to set the frequency of two-way radio transceivers for
- Today, crystals are associated with a counter and holding register on a digital computer.

1960s ERA radio crystal →

- Each oscillation decrements a counter by one
- When counter gets to zero, an interrupt fires
- Can program timer to generate interrupt, let's say 60 times a second, or another frequency to track time

COMPUTER CLOCKS



- Digital clock on computer sets base time
- Crystal clock tracks forward progress of time
  - Translation of wave "ticks" to clock pulses
- CMOS battery on motherboard maintains clock on power loss
- **Clock skew**: physical clock crystals are not exactly the same
- Some run at slightly different rates
- Time differences accumulate as clocks drift forward or backward slightly
- In an automobile, where there is no clock synchronization, clock skew may become noticeable over months, years

UNIVERSAL COORDINATED TIME

- **Universal Coordinated Time (UTC)**

```
ubuntu@ip-172-31-58-89:~$ date
Thu Nov 16 10:13:39 UTC 2017
```

  - Worldwide standard for time keeping
  - Equivalent to Greenwich Mean Time (United Kingdom)
  - 40 shortwave radio stations around the world broadcast a short pulse at the start of each second (WWV)
  - World wide "atomic" clocks powered by constant transitions of the non-radioactive caesium-133 atom
    - 9,162,631,770 transitions per second
- Computers track time using UTC as a base
  - Avoid thinking in local time, which can lead to coordination issues
  - Operating systems may translate to show local time

COMPUTING: CLOCK CHALLENGES

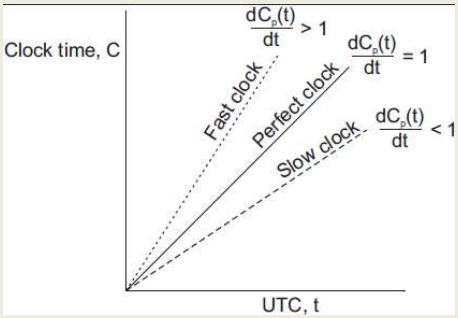
- How do we synchronize computer clocks with real-world clocks?
- How do we synchronize computer clocks with each other?

CLOCK SYNCHRONIZATION

- **UTC services**: use radio and satellite signals to provide time accuracy to 50ns
- **Time servers**: Server computers with UTC receivers that provide accurate time
- **Precision** (π): how close together a set of clocks may be
- **Accuracy**: how correct to actual time clocks may be
- **Internal synchronization**: Sync local computer clocks
- **External synchronization**: Sync to UTC clocks
- **Clock drift**: clocks on different machines gradually become out of sync due to crystal imperfections, temperature differences, etc.
- **Clock drift rate**: typical is 31.5s per year
- **Maximum clock drift rate** (ρ): clock specifications include one

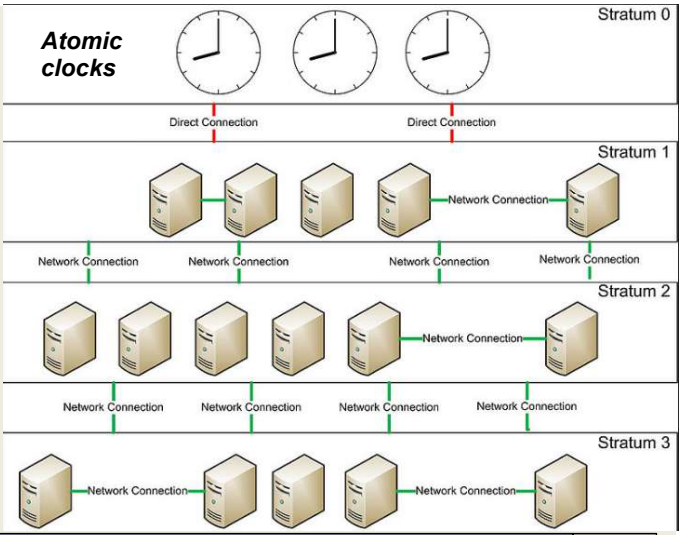
CLOCK SYNCHRONIZATION - 2

- If two clocks drift from UTC in opposite directions, after time Δt after synchronization, they may be 2ρ apart.
- Clocks must be resynchronized every $\pi/2\rho$ seconds
- **Network time protocol**
- Provide coordination of time for servers
- Leverage distributed network of time servers



NETWORK TIME PROTOCOL

- Servers organized into stratums
- Stratum-1 servers have UTC receivers and are sync'd with atomic clocks
- Servers connect with closest NTP server for time synchronization
- Servers assume role as NTP server at stratum+1

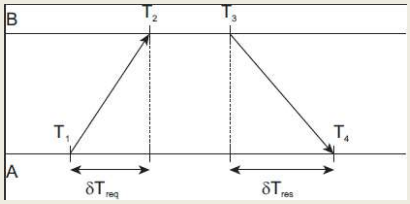


NTP - 2

- Must estimate network delays when synchronizing with remote UTC receiver clocks / time servers

Time server B

Client A



1. A sends message to B, with timestamp T1
2. B records time of receipt T2 (from local clock)
3. B returns response with send time T3, and receipt time T2
4. A records arrival of T4

- Assuming propagation delay of A→B→A is the same
- Estimate propagation delay:
- Add delay to time

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

NTP - 3

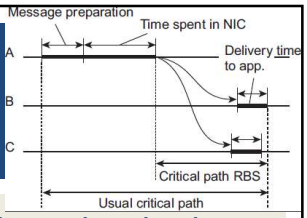
- Cannot set clocks backwards (recall "make" file example)
- Instead, temporarily slow the progress of time to allow fast clock to align with actual time
- Change rate of clock interrupt routine
- Slow progress of time until synchronized
- NTP accuracy is within 1-50ms

- In Ubuntu Linux, to quickly synchronize time:
  `$apt install ntp ntpdate`
- Specify local timeservers in /etc/ntp.conf
  ```
  server time.u.washington.edu iburst
  server bigben.cac.washington.edu iburst
  ```
- Shutdown service (sudo service ntp stop)
- Run ntpdate: (sudo ntpdate time.u.washington.edu)
- Startup service (sudo service ntp start)

BERKELEY ALGORITHM

- Berkeley time daemon server actively polls network to determine average time across servers
- Suitable when no machine has a UTC receiver
- Time daemon instructs servers how much to adjust clocks to achieve precision
- Accuracy can not be guaranteed
- Berkeley is an internal clock synchronization algorithm

CLOCK SYNCHRONIZATION IN WIRELESS NETWORKS



- Sensor networks bring unique challenges for clock synchronization
  - ***Address resource constraints***: limited power, multihop routing slow
- **Reference broadcast synchronization (RBS)**
- Provides precision of time, not accuracy as in Berkeley
- No UTC clock available
- RBS sender broadcasts a reference message to allow receivers to adjust clocks
- No multi-hop routing
- Time to propagate a signal to nodes is roughly constant
- Message propagation time does not consider time spent waiting in NIC for message to send
  - Wireless network resource contention may force wait before message even ***can*** be sent

REFERENCE BROADCAST SYNCHRONIZATION (RBS)

- Node broadcasts reference message m
- Each node p records time Tp,m when m is received
- Tp,m is read from node p's clock
- Two nodes p and q can exchange delivery times to estimate mutual relative offset
- Then calculate relative average offset for the network:

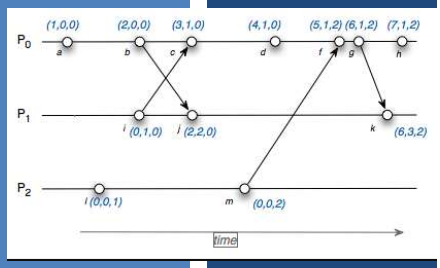
$$Offset[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

- Where M is the total number of reference messages sent
- Nodes can simply store offsets instead of frequently synchronizing clocks to save energy

REFERENCE BROADCAST SYNCHRONIZATION (RBS) - 2

- Cloud skew: over time clocks drift apart
- Averages become less precise
- Elson et al. propose using standard linear regression to predict offsets, rather than calculating them
- IDEA: Use node's history of message times in a simple linear regression to continuously refine a formula with coefficients to predict time offsets:

$$Offset[p,q](t) = \alpha t + \beta$$



CH. 6.2: LOGICAL CLOCKS

L13.63

LOGICAL CLOCKS

- In distributed systems, synchronizing to actual time may not be required…
- It may be sufficient for every node to simply agree on a current time (e.g. logical)
- ***Logical clocks*** provide a mechanism for capturing chronological and ***causal*** relationships in a distributed system
- Think ***counters . . .***
- Leslie Lamport [1978] seminal paper showed that absolute clock synchronization often is not required
- Processes simply need to agree on the order in which events occur

LOGICAL CLOCKS - 2

- **Happens-before relation**
- A→B: **Event A**, happens before **event B**…
- All processes must agree that **event A** occurs first
- Then afterward, **event B**
- Actual time not important. . .

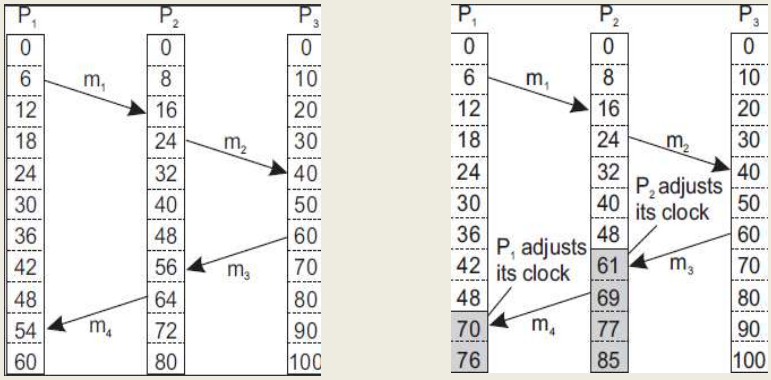
- If **event A** is the event of proc P1 sending a msg to a proc P2, and **event B** is the event of proc P2 receiving the msg, then A→B is also true. . .
- The assumption here is that message delivery takes time
- Happens before is a transitive relation:
- A→B, B→C, therefore A→C

LOGICAL CLOCKS – 3

- If two events, say event X and event Y do not exchange messages, not even via third parties, then X→Y and Y→X **can not be determined**
- Within the system, these events appear **concurrent**
- **Concurrent:** nothing can be said about when the events happened, or which event occurred first
- Clock time, C, must always go forward (increasing), never backward (decreasing)
- Corrections to time can be made by adding a positive value, but never by subtracting one

LOGICAL CLOCKS - 4

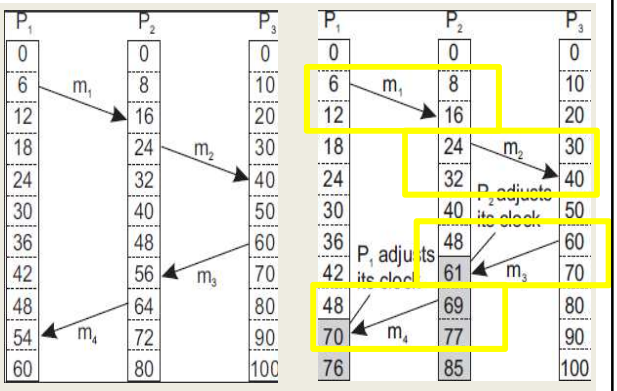
- Three processes each with local clocks
- ***Lamport's algorithm*** corrects their values



LOGICAL CLOCKS

- **Events**:

6: P1 send m1 to P2
16: P2 receives m1
24: P2 sends m2 to P3
40: P3 receives m2
60: P3 sends m3 to P2
56: P2 receives m3
56: P2 clock reset=61
64: P2 sends m4 to P1
54: P1 receives m4
70: P1 clock reset=70



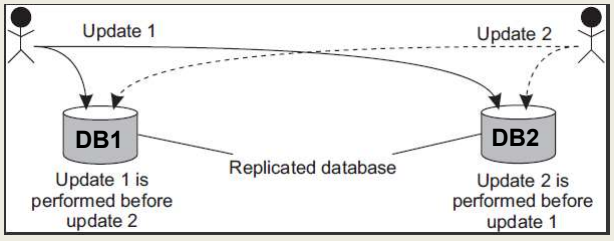
LAMPORT LOGICAL CLOCKS - IMPLEMENTATION

- Negative values not possible
- When a message is received, and the local clock is before the timestamp when then message was sent, the local clock is updated to message_sent_time + 1

1. Clock is incremented before an event: sending a message, receiving a message, some other internal event
   Pi increments Ci: Ci ← Ci + 1
2. When Pi send msg m to Pj, m's timestamp is set to Ci
3. When Pj receives msg m, Pj adjusts its local clock
   Cj ← max{Cj, ts(m)}
4. Ties broken by considering Proc ID: i<j; <40,i> < <40,j>

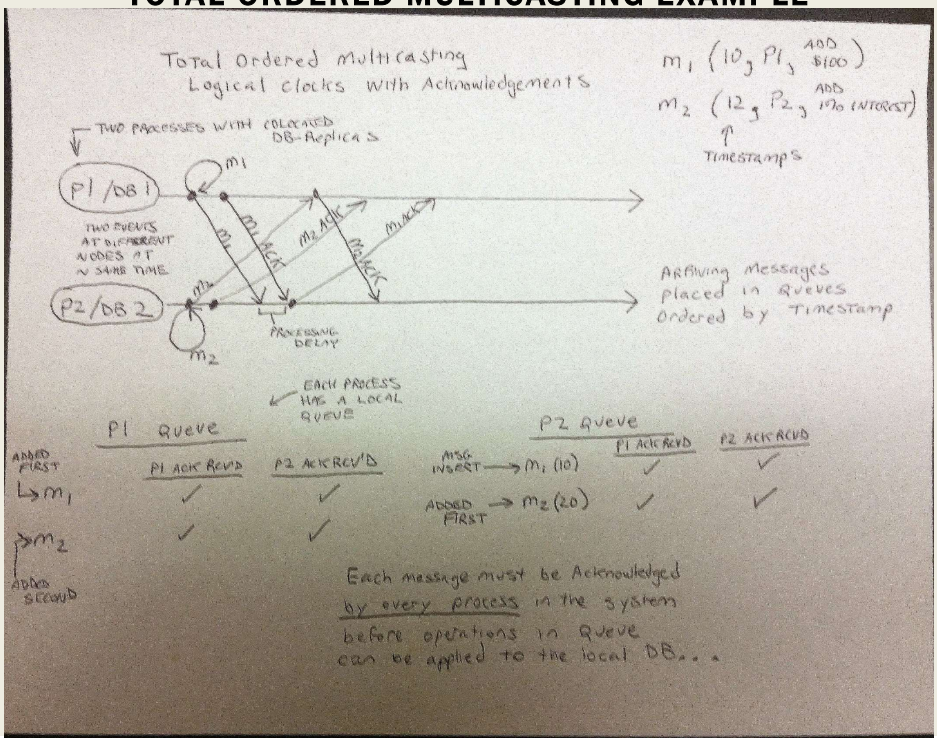
TOTAL-ORDERED MULTICASTING

- Consider concurrent updates to a replicated database
- Communication latency between DB1 and DB2 is 250ms



- **Initial Account balance**: $1,000
- **Update #1**: Deposit $100
- **Update #2**: Add 1% Interest
- Total Ordered Multicasting needed

TOTAL-ORDERED MULTICASTING EXAMPLE

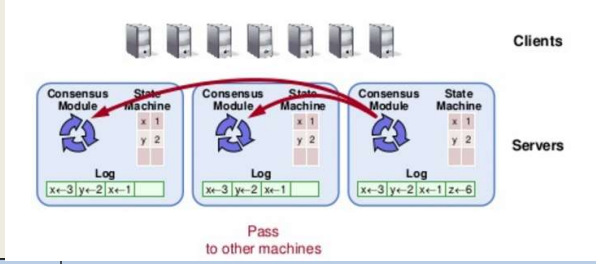


TOTAL-ORDERED MULTICASTING - 2

- Each message timestamped with local logical clock of sender
- Multicast message is conceptually sent to the sender
- Assumptions:
  - Messages from same sender received in order they were sent
  - No messages are lost
- When messages arrive they are placed in local queue ordered by timestamp
- Receiver multicasts acknowledgement of message receipt to other processes
  - Time stamp of message receipt is lower the acknowledgement
- This process ***replicates*** queues across sites
- Process delivers messages to application only when message at the head of the queue has been acknowledged by every process in the system

TOTAL-ORDERED MULTICASTING - 3

- Can be used to provide replicated state machines (RSMs)
- Concept is to replicate event queues at each node
- (1) ***Using logical clocks*** and (2) ***exchanging acknowledgement messages,*** allows for events to be ***"totally"*** ordered in replicated event queues
- Events can be applied **"in order"** to each (distributed) replicated state machine (RSM)

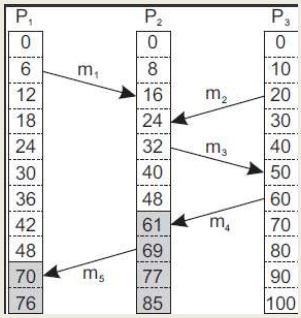


VECTOR CLOCKS

- Lamport clocks don't help to determine causal ordering of messages
- Vector clocks capture causal histories and can be used as an alternative
- What is causality?

WHAT IS CAUSALITY?

- Consider the messages:



- P2 receives m1, and subsequently sends m3
- **Causality:** Sending m3 *may* depend on what's contained in m1
- P2 receives m2, receiving m2 is ***not*** related to receiving m1
- ***Is sending m3 causally dependent on receiving m2?***

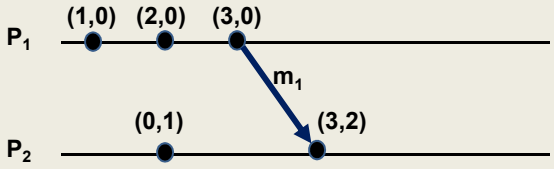
VECTOR CLOCKS

- Vector clocks keep track of **causal history**
- If two local events happened at process P, then the causal history H(p2) of event p2 is {p1,p2}

- P sends messages to Q (*event p3*)
- Q previously performed event q1
- Q records arrival of message as q2
- Causal histories merged at Q H(q2)= {p1,p2,p3,q1,q2}

- Fortunately, can simply store history of last event, as a vector clock → H(q2) = (3,2)
- Each entry corresponds to the last event at the process

VECTOR CLOCKS - 2



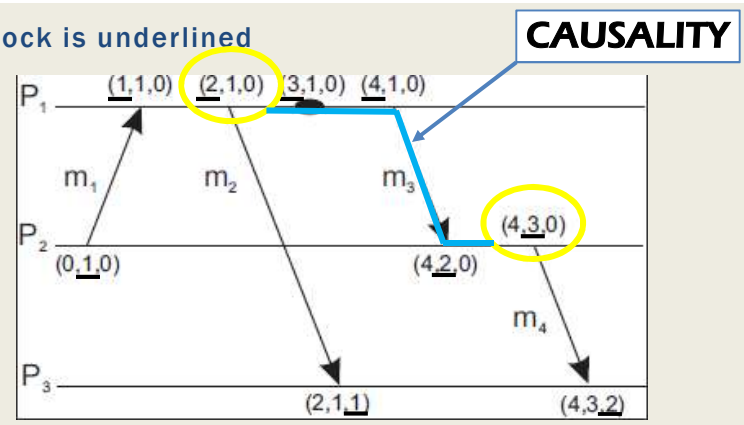
- Each process maintains a vector clock which
  - Captures number of events at the local process (e.g. logical clock)
  - Captures number of events at all other processes
- Causality is captured by:
  - For each event at Pi, the vector clock (VC_i) is incremented
  - The msg is timestamped with VC_i; and sending the msg is recorded as a new event at P_i
  - P_j adjusts its VC_j choosing the **max** of: the message timestamp –or- the local vector clock (VC_j)

VECTOR CLOCKS - 3

- Pj knows the # of events at Pi based on the timestamps of the received message
- Pj learns how many events have occurred at other processes based on timestamps in the vector
- These events ***"may be causally dependent"***
- **In other words:** they may have been necessary for the message(s) to be sent…

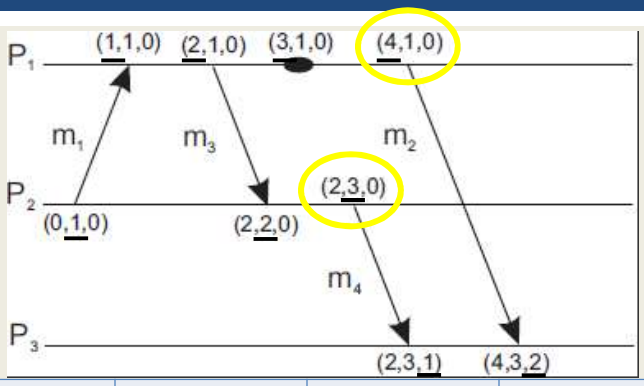
VECTOR CLOCKS EXAMPLE

- Local clock is underlined



ts (m_2)	ts(m_4)	ts(m_2)<ts(m_4)	ts(m_2)>ts(m_4)	Conclusion
(2,1,0)	(4,3,0)	Yes	No	m2 may causally precede m4

VECTOR CLOCKS EXAMPLE - 2

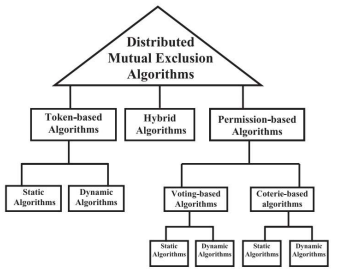


ts (m_2)	ts(m_4)	ts(m_2)<ts(m_4)	ts(m_2)>ts(m_4)	Conclusion
(4,1,0)	(2,3,0)	No	No	m2 and m4 may conflict

- P3 can't determine if m4 may be causally dependent on m2
- **Is m4 causally dependent on m3 ?**

VECTOR CLOCKS - 4

- **Disclaimer:**
- Without knowing actual information contained in messages, it is not possible to state with certainty that there is a causal relationship or perhaps a conflict
- Vector clocks can help us suggest possible causality
- We never know for sure…



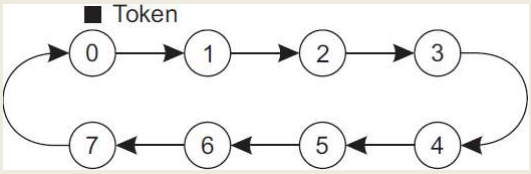
CH. 6.3: DISTRIBUTED MUTUAL EXCLUSION

DISTRIBUTED MUTUAL EXCLUSION

- Coordinating access among distributed processes to a shared resource requires **Distributed Mutual Exclusion**
- **Token-based algorithms:**
- Mutual exclusion by passing a "token" between nodes
- Nodes often organized in ring
- Only one token, holder has access to shared resource
- Avoids starvation: ***everyone gets a chance to obtain lock***
- Avoids deadlock: easy to avoid

TOKEN-RING ALGORITHM

- Construct overlay network
- Establish logical ring among nodes



- Single token circulated around the nodes of the network
- Node having token can access shared resource
- If no node accesses resource, token is constantly circulated around ring

TOKEN-RING CHALLENGES

1. If token is lost, token must be regenerated
   - **Problem**: may accidentally circulate multiple tokens
2. Hard to determine if token is lost
   - What is the difference between token being lost and a node holding the token for a long time?
3. When node crashes, circular network route is broken
   - Dead nodes can be detected by adding a receipt message for when the token passes from node-to-node
   - When no receipt is received, node assumed dead
   - Dead process can be "jumped" in the ring

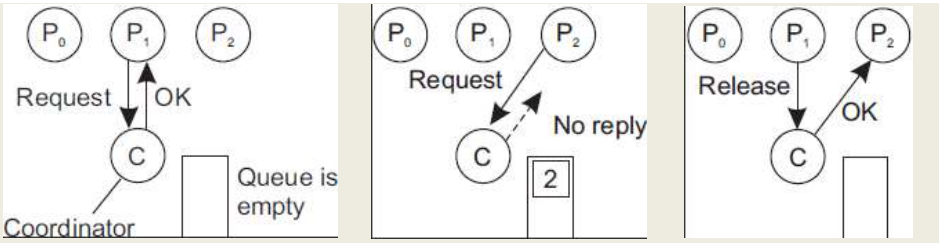
DISTRIBUTED MUTUAL EXCLUSION - 2

- **Permission-based algorithms**
- Processes must require permission from other processes before first acquiring access to the resource

- **Centralized algorithm**
- Elect a single leader node to coordinate access to shared resource(s)
- Manage mutual exclusion on a distributed system similar to how it mutual exclusion is managed for a single system
- Nodes must all interact with leader to obtain ***"the lock"***

CENTRALIZED MUTUAL EXCLUSION

Permission granted from coordinator \/ No response from coordinator



P_1 executes | P_2 blocks | P_1 finishes; P_2 executes

- When resource not available, coordinator can block the requesting process, or respond with a reject message
- P2 must ***poll*** the coordinator if it responds with reject otherwise can wait if simply blocked
- Requests granted permission fairly using FIFO queue
- Just three messages: (request, grant, release)

CENTRALIZED MUTUAL EXCLUSION - 2

- **Issues**
- Coordinator is a single point of failure
- Processes can't distinguish dead coordinator from "permission denied"
  - No difference between CRASH and Block (*for a long time*)
- Large systems, coordinator becomes performance bottleneck
  - Scalability: Performance does not scale

- **Benefits**
- Simplicity:
  Easy to implement compared to distributed alternatives

DISTRIBUTED ALGORITHM

- Ricart and Agrawala [1981], use total ordering of all events
  - Leverages Lamport logical clocks
- Package up resource request message (AKA Lock Request)
- Send to all nodes
- Include:
  - Name of resource
  - Process number
  - Current (logical) time
- Assume messages are sent reliably
  - No messages are lost

DISTRIBUTED ALGORITHM - 2

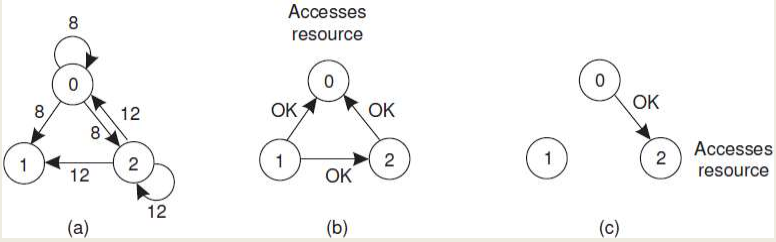
- When each node receives a request message they will:

1. Say OK (***if the node doesn't need the resource***)
2. Make no reply, queue request (***node is using the resource)***
3. Perform a timestamp comparison (***if node is waiting to access the resource***), then:
   1. Send OK if requester has lower logical clock value
   2. Make no reply if requester has higher logical clock value

- Nodes sit back and wait for all nodes to grant permission
- Requirement: every node must know the entire membership list of the distributed system

DISTRIBUTED ALGORITHM - 3

- If Node 0 and Node 2 simultaneously request access
- Node 0's time stamp is lower (8) than Node 2 (12)
- Node 1 and Node 2 grant Node 0 access
- Notice that Node 1 also grants Node 2 permission



- **In case of conflict, lowest timestamp wins!**

CHALLENGES WITH DISTRIBUTED ALGORITHM

- **Problem:** Algorithm has N points of failure !
- Where N = Number of Nodes in the system

- **Problem:** When node is accessing the resource, it does not respond
  - Lack of response can be confused with **failure**
  - Solution: When node receives request for resource it is accessing, always send a reply either granting or denying permission (ACK)
  - Enables requester to determine when nodes have died

CHALLENGES WITH DISTRIBUTED ALGORITHM - 2

- **Problem**: Multicast communication required –or- each node must maintain full group membership
  - Track nodes entering, leaving, crashing…
- **Problem**: Every process is involved in reaching an agreement to grant access to a shared resource
  - This approach ***may not scale*** on resource-constrained systems
- Solution: Can relax total agreement requirement and proceed when a **simple majority** of nodes grant permission
  - *Presumably any one node locking the resource prevents agreement*
- Distributed algorithm for mutual exclusion works best for:
  - Small groups of processes
  - When memberships rarely change

QUESTIONS

EXTRA SLIDES

95