Parallel and Distributed Computing

CS439: Principles of Computer Systems

April 20, 2014

Last Time

Network Programming:

- Client-Server Transactions
- Ports
  - Well-known, ephemeral
- Sockets
  - End point of communication
- Client-Side Programming
  - socket, connect
- Server-Side Programming
  - socket, bind, listen, accept
- Remote Procedure Calls (RPC)

Today's Agenda

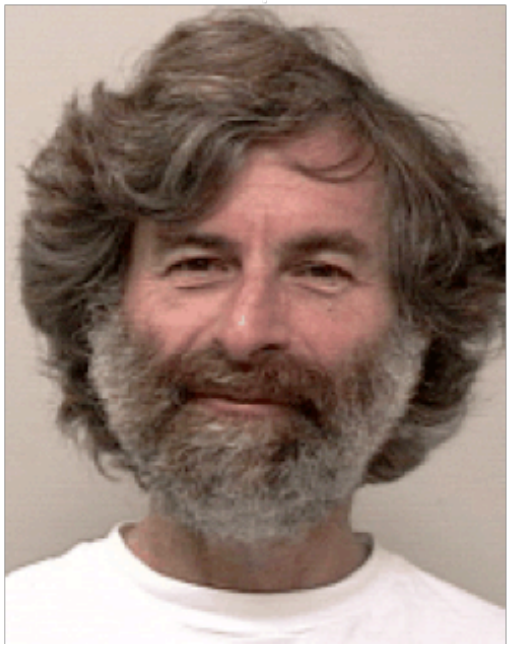
- Parallel and Distributed Computing
  - What
- Parallel Programming Models
  - Shared Memory
  - Message Passing
- Distributed Computing
  - Event Ordering
  - Atomicity of Transactions
    - Two Phase Commit (2PC)

Parallel and Distributed Computing

What is a Distributed System?

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport

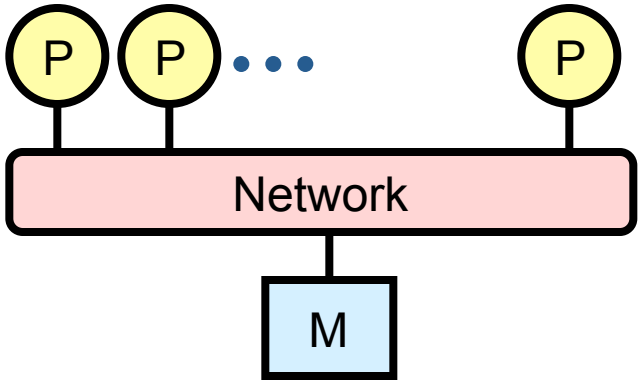


Classification

- A *distributed system* is a set of physically separate processors connected by one or more communication links
- *Parallel Computing*: tightly-coupled systems
  - Processors share clock, memory, and run one OS
  - Frequent communication
- *Distributed Computing*: loosely-coupled systems
  - Each processor has its own memory
  - Each processor runs an independent OS
  - Communication should be less frequent than in parallel computing
  - Supercomputers, Clusters, Massively Parallel Machines
  - Email, File servers, Printer access, Backup over network, WWW

Parallel Computing

- Several processors share one address space
- Communication through shared memory (typically)
  - Read and write accesses to shared memory locations



Conceptual Model

Parallel Computing: Plain Text

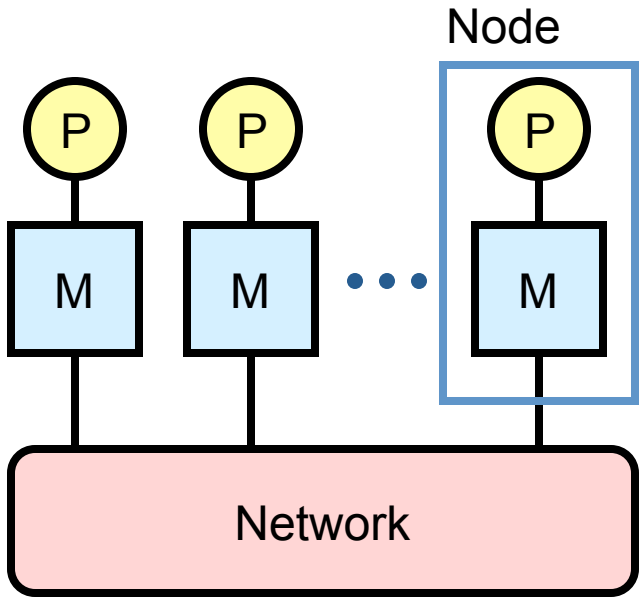
- Several processors share one address space
  - Conceptually a shared memory
- Communication is through shared memory (typically)
  - Read and write accesses to shared memory locations
- Processors connected via a network (typically a bus)
- Network is connected to single shared memory

The Architecture of Parallel Computing

- Two most prevalent:
  - SMP (Symmetric Multiprocessor):
    - *Multiprocessor*: two or more processors have a common RAM
    - *Symmetric* refers to the OS
      - One OS for all processors
      - Any processor can run it
  - Multicore: multiprocessors on the same chip
- Also combined
  - Each blade of Stampede has 2 8-core processors as an SMP unit and 1 61-core co-processor

Distributed Computing

- A collection of computers (nodes) connected by a network
- Communication through message passing



Distributed Computing: Plain Text

- A collection of computers (nodes) connected by a network
- Communication through message passing
- A node is a processor that is connected memory
- The nodes are then linked via the network

The Architecture of Distributed Computing

- Nodes connected by a network
- Massively Parallel Machines
  - Nodes are greatly simplified and include only memory, processor(s), network card
  - Augmented with fast network interface
- Clusters
  - Network workstations with a fast network
    - Built of Common Off-The-Shelf (COTS) parts
  - Less specialized

Very Distributed Computing

- Grid computing
  - Multiple locations
  - Heterogeneous architectures
  - Example: XSEDE
- Cloud computing

Parallel Programming
(includes parallel and distributed architectures)

Why a different type of programming?

Sequential programs get no benefit from multiple processors

- Key property is how much communication per unit of computation.
  - The less communication per unit computation the better the scaling properties of the algorithm.
- Sometimes, a multi-threaded design is good on uni- and multi-processors
  - e.g., throughput for a web server (that uses kernel threading)
- Many applications can be (re)designed/coded/ compiled to generate cooperating, parallel instruction streams
  - Improving responsiveness/throughput with multiple processors.

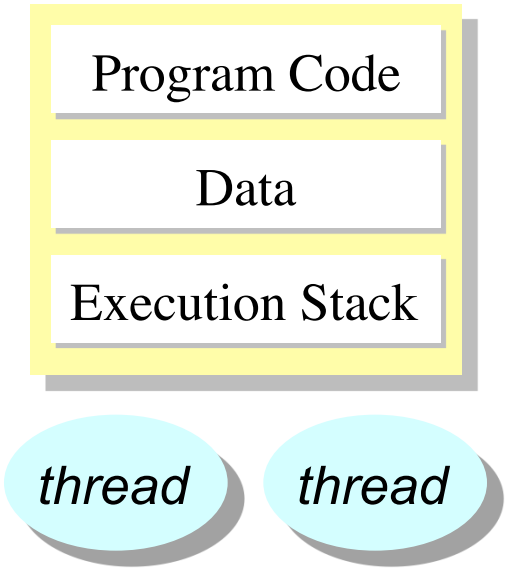
Parallel Programming

- Parallel programming involves:
  - Decomposing an algorithm into parts
  - Distributing the parts as tasks which are worked on by multiple processors simultaneously
  - Coordinating work and communication of those processors
    - Synchronization
- Parallel programming considerations:
  - Type of parallel architecture being used
  - Type of processor communications used
- No automated compiler/language exists to automate this “parallelization” process.

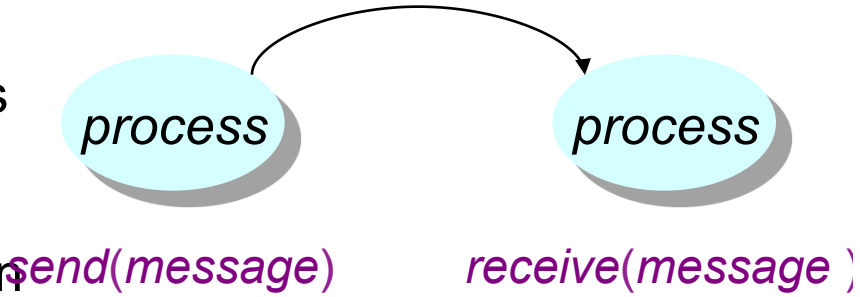
Parallel Programming Models

Communication and synchronization based on either:

- Shared memory
  - Interprocess communication is implicit
  - Synchronization is explicit
  - Assume processes/threads can read & write a set of shared memory locations
  - Difficult to provide across machine boundaries



- Message passing
  - Interprocess communication is explicit
  - Synchronization is implicit
  - Extensible to communication in distributed systems



Parallel Programming Models: Plain Text

- Communication and synchronization based on either:
  - Shared memory
    - Interprocess communication is implicit
    - Synchronization is explicit
    - Assume processes/threads can read & write a set of shared memory locations
    - Difficult to provide across machine boundaries
    - Example:
      - One process with program code, data, execution stack
      - Two threads
  - Message passing
    - Interprocess communication is explicit
    - Synchronization is implicit
    - Extensible to communication in distributed systems
    - Example:
      - Two processes
      - Process A sends the message: send(message)
      - Process B receives the message: receive(message)

Shared Memory Programming Model

- Programs/threads communicate/cooperate via loads/stores to memory locations they share.
- Communication is therefore at memory access speed (very fast) and is implicit.
- Cooperating pieces must all execute on the same system (computer).
- OS services and/or libraries used for creating tasks (processes/threads) and coordination (semaphores/barriers/locks.)

Message Passing Programming Model

- “Shared” data is communicated using send/receive services (across an external network).
- Shared data must be formatted into message chunks for distribution
  - Unlike shared memory
- Coordination is also via sending/receiving messages.
- Program components can be run on the same or different systems, so can use many of processes.
- Standard libraries exist to encapsulate messages:
  - Parasoft's Express (commercial)
  - PVM (standing for Parallel Virtual Machine, non-commercial)
  - MPI (Message Passing Interface, also non-commercial).

Message Passing Logistics: Synchronization

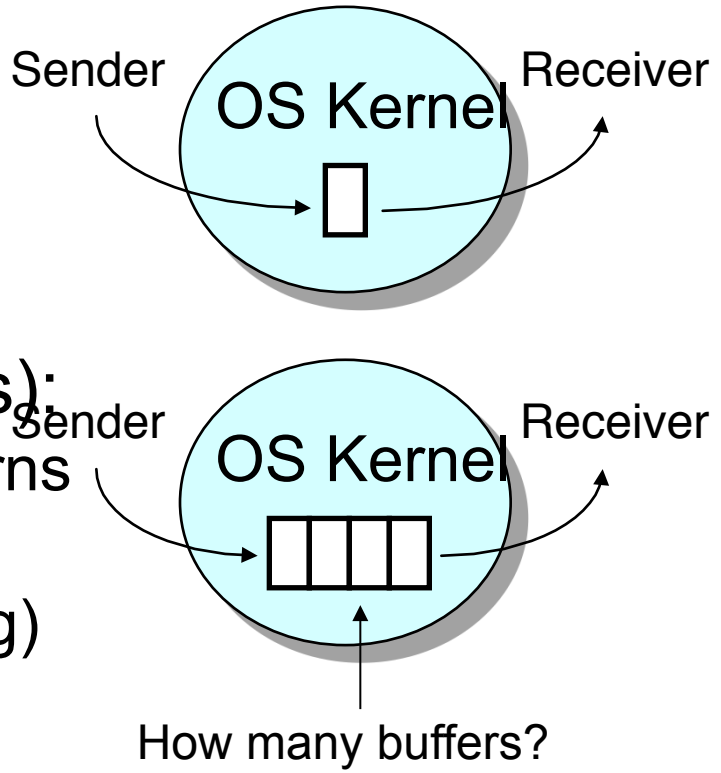
When do *send*/*receive* operations terminate?

Blocking (aka Synchronous):

- Sender waits until its message is received
- Receiver waits if no message is available

Non-blocking (aka Asynchronous):

- Send operation "immediately" returns
- Receive operation returns if message is available or not (polling)



Partially blocking/non-blocking:

- send()/receive() with *timeout*

Message Passing Logistics: Synchronization Plain Text

- When does a send/receive operation terminate?
  - Blocking (aka Synchronous):
    - Sender waits until its message is received
    - Receiver waits if no message is available
  - Non-blocking (aka Asynchronous):
    - Send operation “immediately” returns
    - Receive operation returns if message is available or not (polling)
  - Partially blocking/non-blocking
    - send()/ receive() with timeout
- The OS kernel has a buffer(s) to hold messages between the sender and the receiver
  - For blocking sends and receives, only one buffer is needed
  - For non-blocking, more than one is potentially needed, and the number of buffers is a design decision

Limitations of Message Passing

Easy for OS, hard for programmer

- Programmer must code communication
- Programmer may have to code format conversions, flow control, error control
- No dynamic resource discovery

Event Ordering

Event Ordering

- Coordination of requests (especially in a fair way) requires events (requests) to be ordered.

- Stand-alone systems:
  - Shared Clock / Memory
  - Use a time-stamp to determine ordering
- Distributed Systems: What does time mean?
  - No global clock
  - Each clock runs at different speeds (clock drift)

- How do we order events running on physically separated systems?

Event Ordering: Distributed Systems

- Through message passing
- A message must be sent before it can be received
- Send/Receives can thus “synchronize” the clocks

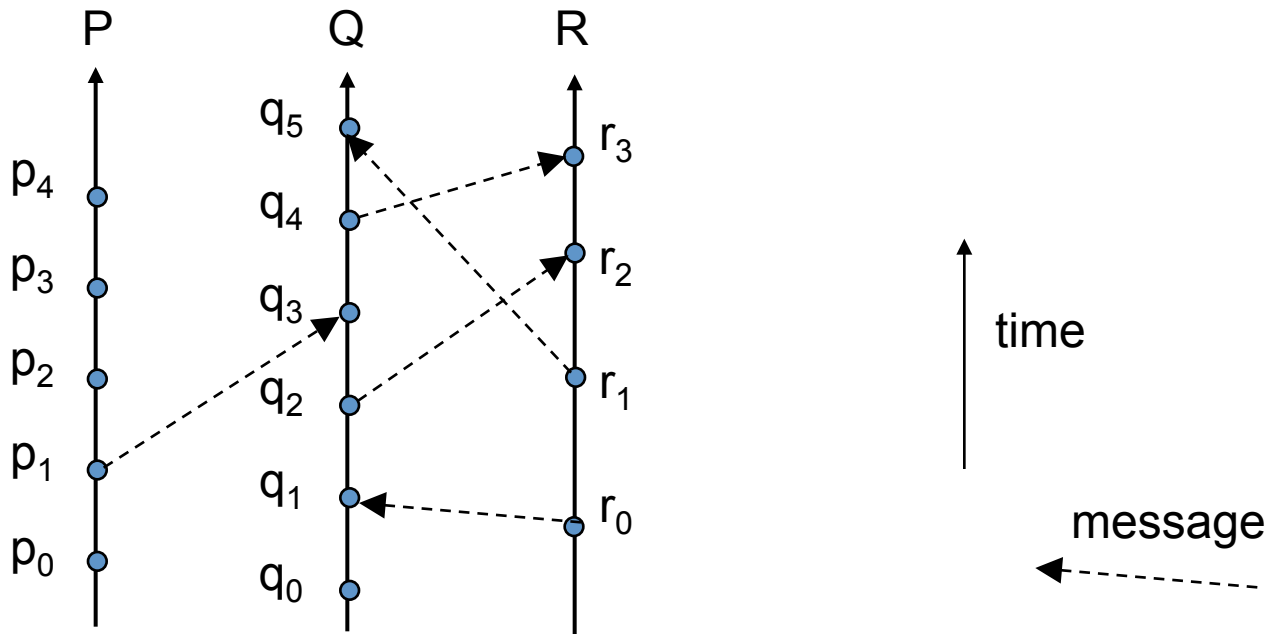
Event Ordering: Happened-Before Relation

1. If A and B are events in the same process, and A executed before B, then $A \rightarrow B$.

2. If A is a message send and B is when the message is received, then $A \rightarrow B$.

3. $A \rightarrow B$, and $B \rightarrow C$, then $A \rightarrow C$

Happened-Before Relationship



Ordered events

p_1 precedes ___

q_4 precedes ___

q_2 precedes ___

p_0 precedes ___

Unordered (Concurrent) events

q_0 is concurrent with ___

q_2 is concurrent with ___

q_4 is concurrent with ___

q_5 is concurrent with ___

Happened-Before Relationship

- We are mapping the events of 3 different processes, P, Q, and R, over time
- For each process the events happen in numerical order
  - E.g., p0 happens before p1 which happens before p2, etc.
- We also map the messages sent by processes at certain events and received by others in order to determine the relative order of different events
- Events by process:
  - Process P: p0, p1, p2, p3 and p4
  - Process Q: q0, q1, q2, q3, q4 and q5
  - Process R: r0, r1, r2 and r3
- Messages between processes' events:
  - p1 sent a message to q3 (as in Process P sends a message to Process Q at event p1 and it is received at Q's event q3)
  - q2 sent a message to r2
  - q4 sent a message to r3
  - r0 sent a message to q1
  - r1 sent a message to q5
- Ordered events (fill in the blank)
  - p1 precedes
  - q4 precedes
  - q2 precedes
  - p0 precedes
- Unordered (concurrent) events (fill in the blank)
  - q0 is concurrent with
  - q2 is concurrent with
  - q4 is concurrent with
  - q5 is concurrent with

Happened Before and Total Event Ordering

Define a notion of event ordering such that:

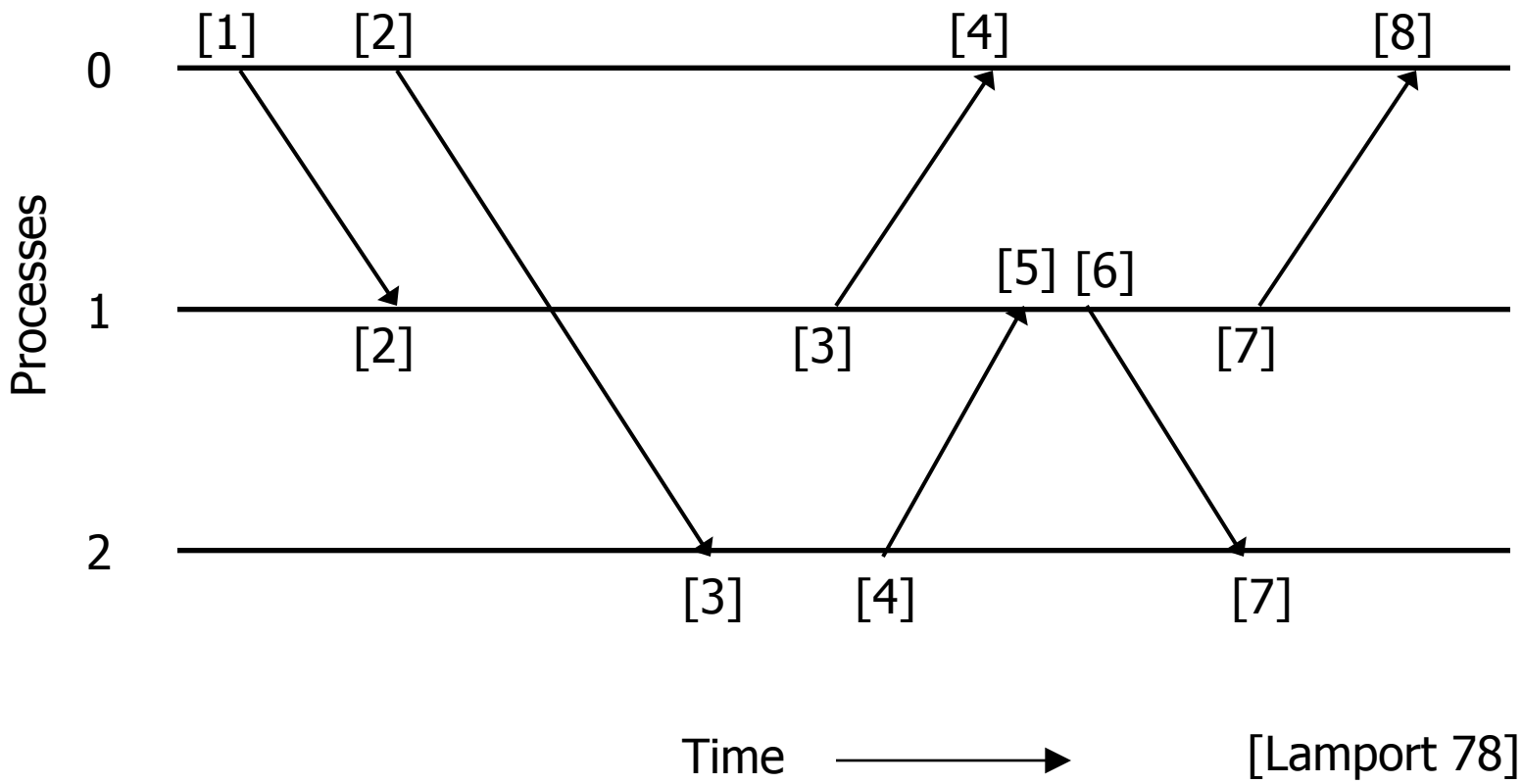
1. If A → B, then A precedes B.
2. If A and B are concurrent events, then nothing can be said about the ordering of A and B.

Solution:

1. Each process i maintains a *logical clock,* LC_i
2. When an event occurs locally, $LC_i = LC_i + 1$
3. When process X sends a message to Y, it also sends a *time stamp* LC_x in the message.
4. When Y receives this message, it:

```
LC_y = LC_y + 1
if LC_y < (LC_x + 1) //if Y's clock is behind X's
    LC_y  = LC_x + 1; //set Y's to X's value + 1
```

Logical Clocks: Example



Logical Clocks: Example
Plain Text

- We have 3 different processes 0, 1, and 2, and we are determining their logical clock values at each send/receive
  - In this scenario, we are only concerned with the send and receive events---there are no other local events
- Different clock values indicate that the events must happen in that order based on their happened-before relationships
- Order of events in the system:
  1. Process 0 sends a message to Process 1
     - Send is Event 1 on Process 0
     - Receive is Event 2 on Process 1
     - Process 1 does not have an Event 1. Its logical clock's initial value is set according to Process 0's and the rules of logical clocks.
  2. Process 0 sends a message to Process 2
     - Send is Event 2 on Process 0
     - Receive is Event 3 on Process 2
     - Process 2 does not have Events 1 or 2. Its logical clock's initial value is set according to Process 0's and the rules of logical clocks.
  3. Process 1 sends a message to Process 0
     - Send is Event 3 on Process 1
     - Receive is Event 4 on Process 0
     - Process 0 does not have an Event 3. Its logical clock is updated according to the value of Process 1's clock and the rules of logical clocks.
  3. Process 2 sends a message to Process 1
     - Send is Event 4 on Process 2
     - Receive is Event 5 on Process 1
     - Process 1 does not have an Event 4. Its logical clock is updated according to the value of Process 2's clock and the rules of logical clocks.
  4. Process 1 sends a message to Process 2
     - Send is Event 6 on Process 1
     - Receive is Event 7 on Process 2
     - Process 2 does not have Events 5 or 6. Its logical clock is updated according to the value of Process 1's clock and the rules of logical clocks.
  5. Process 1 sends a message to Process 0
     - Send is Event 7 on Process 1
     - Receive is Event 8 on Process 0
     - Process 0 does not have Events 5, 6, or 7. Its logical clock is updated according to the value of Process 1's clock and the rules of logical clocks.

iClicker Question

If A→B and C→B can we say A→C?

A. Yes
B. No

Atomicity

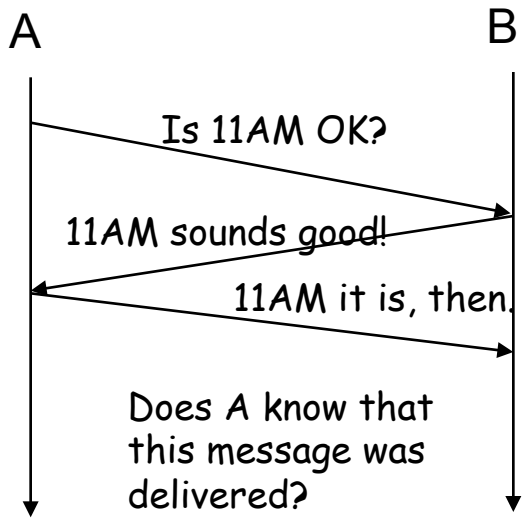
Atomicity in Distributed Systems

- How can we atomically update state on two different systems?
- Examples:
  - Atomically move a file from server A to server B
  - Atomically move $100 from one bank to another
- Issues:
  - Messages exchanged by systems can be lost
  - Systems can crash
- Can we use messages and retries over an unreliable network to *synchronize* the actions of two machines?

The Generals' Paradox

Problem:

- Two generals are on two separate mountains
- Can communicate only via messengers; but messengers can get lost or captured by enemy
- Goal is to coordinate their attack
  - If attack at different times → they lose !
  - If attack at the same time → they win !



Even if all previous messages get through, the generals still can't coordinate their actions, since the last message could be lost, always requiring another confirmation message.

The Generals' Paradox: Plain Text

Problem:

- 2 generals are on 2 separate mountains
- Can communicate only via messages; but messengers can get lost or captured by enemy
- Goal is to coordinate their attack
  - If attack at different times they both lose!
  - If attack at the same time they win!
- BUT how can they be really sure that the messages are being received?
  - Confirmation messages?
    - How do you know the last confirmation message was received? Results in infinite pinging back and forth
    - Even if all previous message get through, the generals still can't coordinate their actions, since the last message could be lost, always requiring another confirmation message

iClicker Question

Distributed consensus in the presence of link failures is:

A. Possible

B. Not possible

Distributed Consensus with Link Failures

- Problem:
  - Take any exchange of messages that solves the general's coordination problem.
  - Take the last message m_n. Since m_n might be lost, but the algorithm still succeeds, it must not be necessary.
  - Repeat until no messages are exchanged.
  - No messages exchanged can't be a solution, so our assumption that we have an algorithm to solve the problem must be wrong.
- Distributed consensus in the presence of link failures is impossible.
  - That is why timeouts are so popular in distributed algorithms.
  - Success can be probable, just not guaranteed in bounded time.

Distributed Transactions

- Two machines agree to do something, or not do it, atomically
  - they are not necessarily agreeing to do it at the same time
- The *two-phase commit protocol* allows coordination under reasonable operating conditions
  - Uses log on each machine to track whether commit happened

Two-Phase Commit Protocol: Phase 1

Phase 1: Coordinator requests a transaction

- Coordinator sends a REQUEST to all participants for that transaction
  - Example: C → S1 : "delete foo from /"
    C → S2 : "add foo to /quux"
- On receiving request, participants perform these actions:
  - Execute the transaction locally
  - Write VOTE_COMMIT or VOTE_ABORT to their local logs
  - Send VOTE_COMMIT or VOTE_ABORT to coordinator

Failure case	Success case
S1 decides OK; writes "rm /foo; VOTE_COMMIT" to log; and sends VOTE_COMMIT S2 has no space on disk; so rejects the transaction; writes and sends VOTE_ABORT	S1 and S2 decide OK and write updates and VOTE_COMMIT to log; send VOTE_COMMIT

Two-Phase Commit Protocol: Phase 1 Plain Text

- Phase 1: coordinator requests a transaction
  - Coordinator sends a REQUEST to all participants for that transaction
    - Example:
      - C to S1: “delete foo from /”
      - C to S2: “add foo to /quux”
  - On receiving request, participants perform these actions
    - Execute the transaction locally
    - Write VOTE_COMMIT or VOTE_ABORT to their local logs
    - Send VOTE_COMMIT or VOTE_ABORT to coordinator
- Success case
  - S1 and S2 decided OK and write updates and VOTE_COMMIT to log; send VOTE_COMMIT
- Failure case:
  - S1 decides OK; writes “rm/foo;VOTE_COMMIT” to log; and sends VOTE_COMMIT
  - S2 has no space on disk; so rejects the transaction; write and sends VOTE_ABORT

Two-Phase Commit Protocol: Phase 2

Phase 2: Coordinator commits or aborts the transaction

- Coordinator decides
  - *Case 1*: Coordinator receives VOTE_ABORT or times-out
    - Coordinator writes GLOBAL_ABORT to log and sends GLOBAL_ABORT to participants
  - *Case 2*: Coordinator receives VOTE_COMMIT from all participants
    - Coordinator writes GLOBAL_COMMIT to log and sends GLOBAL_COMMIT to participants
- Participants commit the transaction
  - On receiving a decision, participants write GLOBAL_COMMIT or GLOBAL_ABORT to log

Does Two-Phase Commit Work?

- Yes … can be proved formally
- Consider the following cases:
  - What if participant crashes during the request phase before writing anything to log?
    - On recovery, participant does nothing; coordinator will timeout and abort transaction; and retry!
  - What if coordinator crashes during phase 2?

    On restart:
    - Case 1: Log does not contain GLOBAL_* → send GLOBAL_ABORT to participants and retry
    - Case 2: Log contains GLOBAL_ABORT → send GLOBAL_ABORT to participants
    - Case 3: Log contains GLOBAL_COMMIT → send GLOBAL_COMMIT to participants

iClicker Question

Can the Two-Phase Commit protocol fail to terminate?

A. Yes

B. No

Limitations of Two-Phase Commit

- What if the coordinator crashes during Phase 2 (before sending the decision) and does not wake up?
  - All participants block forever!
    (They may hold resources – e.g. locks!)

- Possible solution:
  - Participant, on timing out, can make progress by asking other participants (if it knows their identity)
    - If any participant had heard GLOBAL_ABORT → abort
    - If any participant sent VOTE_ABORT → abort
    - If all participants sent VOTE_COMMIT but no one has heard GLOBAL_* → can we commit?
      - NO – the coordinator could have written GLOBAL_ABORT to its log (e.g., due to local error or a timeout)

Two-Phase Commit: Summary

- Message complexity 3(N-1)
  - Request/Reply/Broadcast, from coordinator to all other nodes.
- When you need to coordinate a transaction across multiple machines … Use two-phase commit
  - For two-phase commit, identify circumstances where indefinite blocking can occur
  - Decide if the risk is acceptable
- If two-phase commit is not adequate, then …
  - Use *advanced distributed coordination techniques*
  - To learn more about such protocols, take a distributed computing course

The King has Died!
Long Live the King!

Electing a Leader

Who's in charge?

Many algorithms require a coordinator.

What happens when the coordinator dies (or at startup)?

Let's have an election!

- Bully algorithm

Bully Algorithm

- Assumptions
  - Processes are numbered (otherwise impossible).
  - Using process numbers does not cause unfairness.
- Algorithm concept
  - If leader is not heard from in a while, assume s/he crashed.
  - Leader will be remaining process with highest rank.
  - Processes who think they are leader-worthy will broadcast that information.
  - During this “election campaign” processes who are near the top see if the process trying to grab power crashes (as evidenced by lack of message in timeout interval).
  - At end of time interval, if alpha-process has not heard from rivals, assumes s/he has won.
  - If former alpha-process arises from dead, s/he bullies their way to the top. (Invariant: highest # process rules)

Bully Algorithm Details

- Algorithm starts with P_i broadcasting its desire to become leader. P_i waits T seconds before declaring victory.
- If, during this time, P_i hears from P_j, $j>i$, P_i waits another U seconds before trying to become leader again. U?
  - U ~ 2T, or T + time to broadcast new leader
- If not, when P_i hears from only P_j, $j<i$, and T seconds have expired, then P_i broadcasts that it is the new leader.
- If P_i hears from P_j, $j<i$ that P_j is the new leader, then P_i starts the algorithm to elect itself (P_i is a bully).
- If P_i hears from P_j, $j>i$ that P_j is the leader, it records that fact.

Bully Algorithm: Last Note

When a process recovers and its rank is higher than that of the current leader, it begins the election algorithm and wrestles control away from the current leader.

Summary

- Parallel Computing: tightly-coupled systems
- Distributed Computing: loosely-coupled systems
- Parallel Programming Models
  - Shared Memory (OpenMP)
    - Sharing is implicit, synchronization is explicit
  - Message Passing (MPI)
    - Sharing is explicit, synchronization is implicit
- Distributed systems may need to:
  - have a sense of time: use the *happened before* relationship to order events
  - provide atomicity: use two-phase commit protocol (at some level, the network is unreliable!)

Announcements

- Project 4 (Last one!) due Friday, 5/8, 11:59p
  - No slip days!
- Homework 10 due Friday 8:45a
- If you have a conflict for the final, you should have already contacted me (email, please!)
  - Thursday, May 14, 7p-10p in UTC 2.102A