Parallel and Distributed Computing

Bringing It All Together

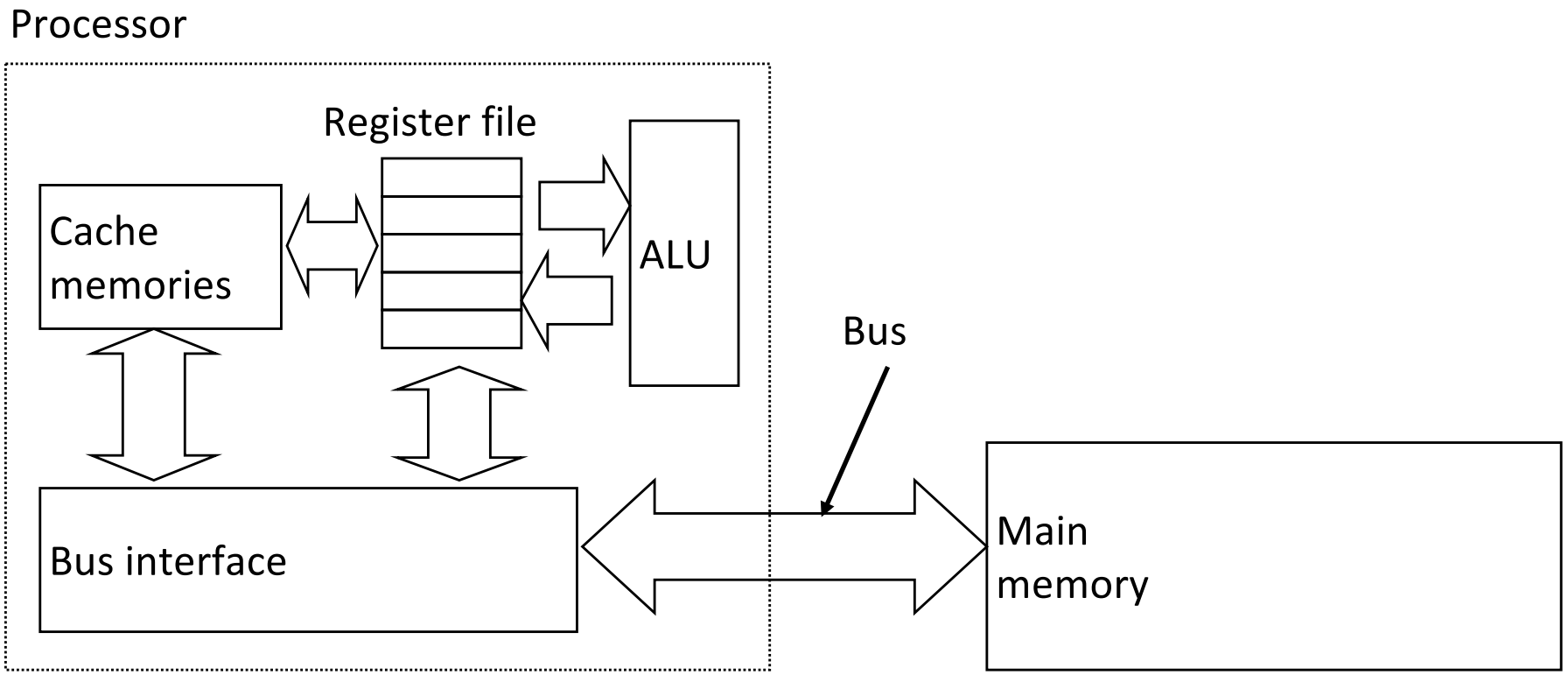
- We've been studying how an OS manages a single CPU system
- As part of that, it will communicate with I/O devices
- The network is one of those I/O devices
  - Networks are controlled by complex software that enables the proper routing of data and the appearance of reliability
  - As a programmer, we access the network through an abstraction called a socket
- Parallel programming techniques are used in both parallel and distributed applications
  - Include shared memory and message passing

Parallel Computing

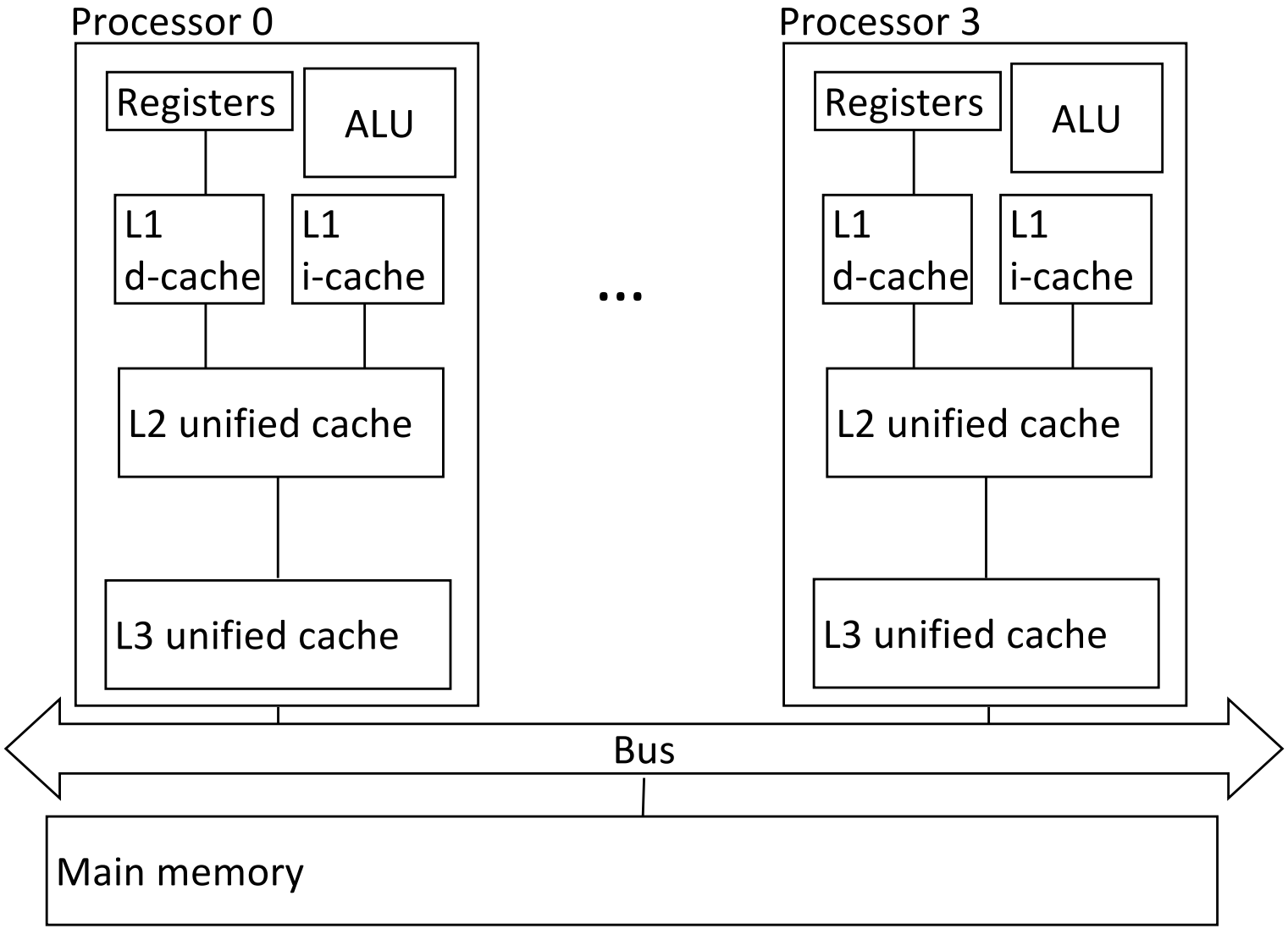
Parallel Computing: Description

- Tightly-coupled systems
- Processors share:
  - Clock
  - Memory (a single physical address space)
  - A single OS
- Examples: Multicore systems, Symmetric Multi Processor (SMP) systems (and you can make SMP multicore processors…)

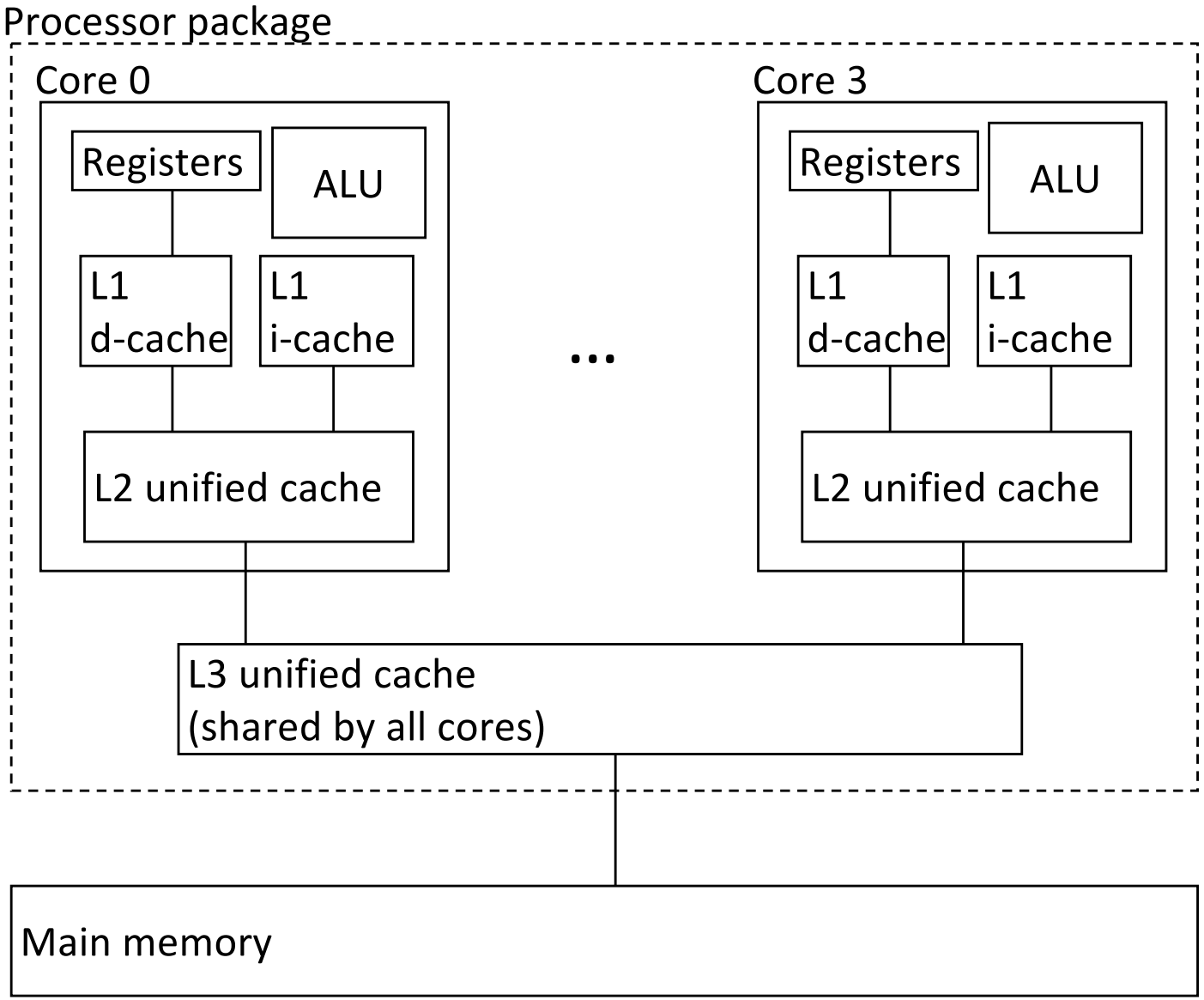
Single CPU


Processor
Register file
Cache memories
ALU
Bus
Bus interface
Main memory

Symmetric Multi Processor



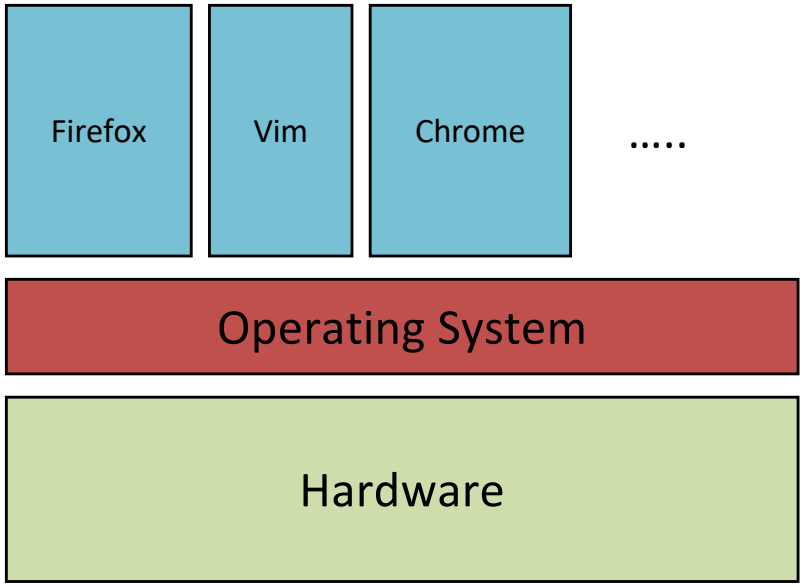
Multicore Processor



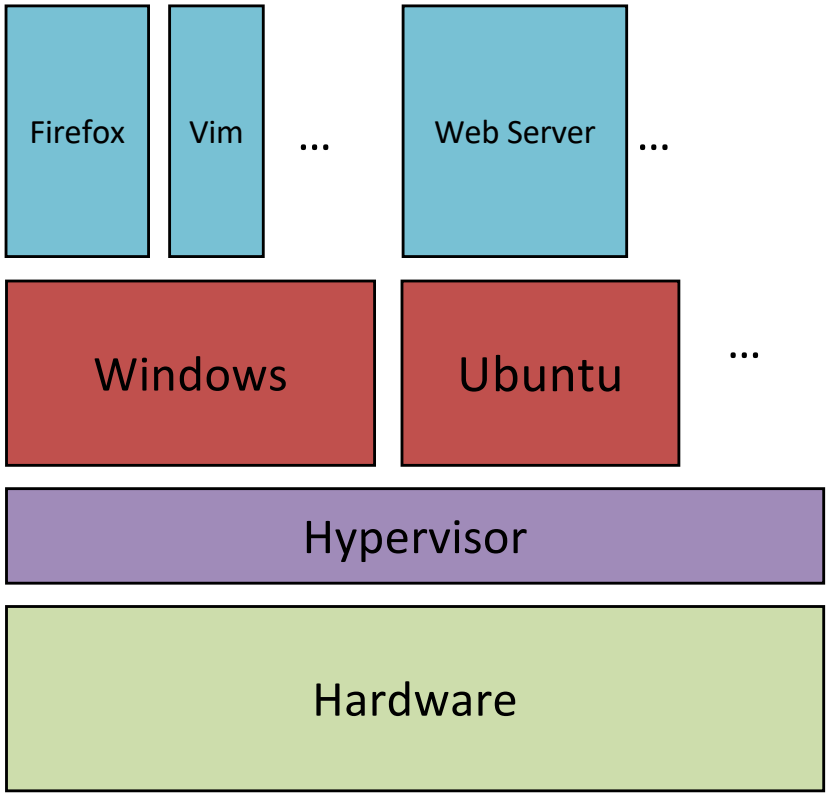
Virtualization

Virtual Machines

Traditional



with Virtualization

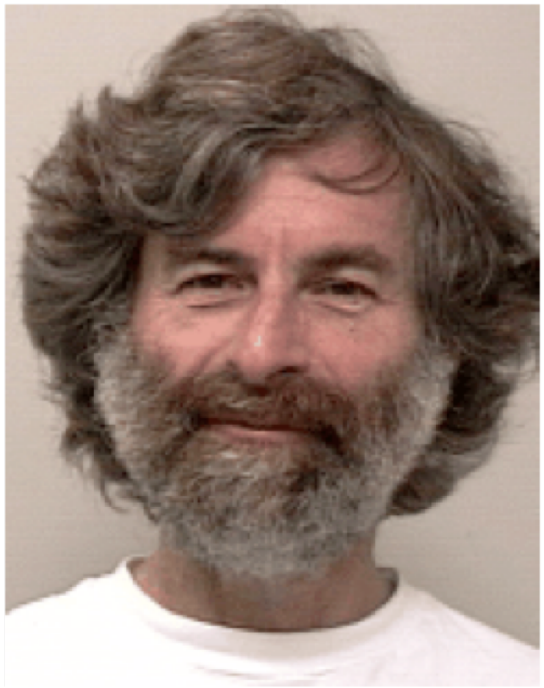


Distributed Computing

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."

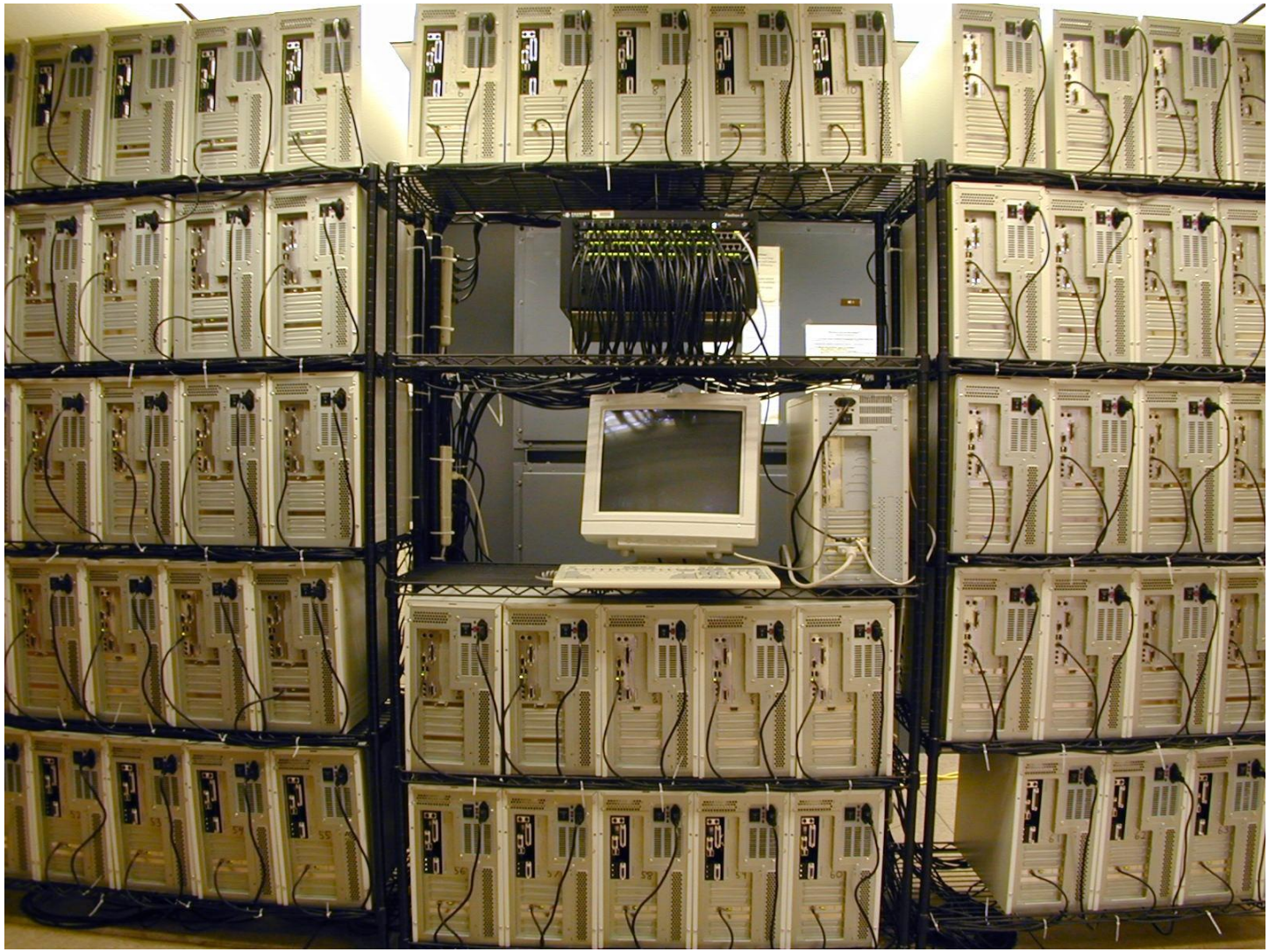
Leslie Lamport



Distributed Computing: Description

- Loosely-coupled systems
- A set of physically separate processors connected by one or more communication links
  - Nodes connected by a network
- As a result:
  - Each processor has its own memory
  - Each processor runs an independent OS
  - Each processor has its own clock
- Communication is more expensive than in parallel computing
- Examples:
  - Email, File servers, Printer access, Backup over network, WWW
  - Supercomputers, Clusters, Massively Parallel Machines

A Cluster



64 dual-processor PCs connected by a 100Mb/Sec Ethernet switch

Frontera: A TACC Supercomputer



Very Distributed Computing

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

Unique Challenges: Event Ordering

Event Ordering

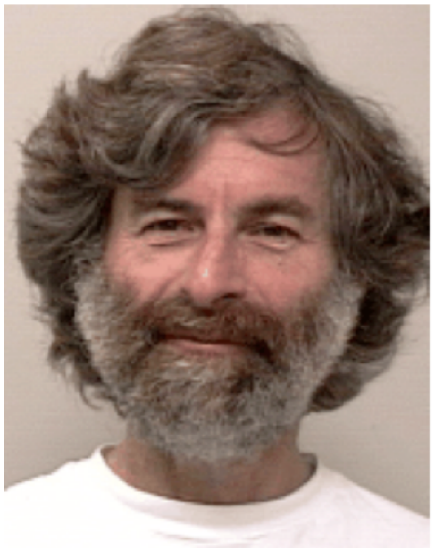
- Coordination of requests (especially in a fair way) requires events (requests) to be ordered.

- Stand-alone systems/Parallel Computing:
  - 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 executing on physically separated systems?
  - *Recall that communication is through message passing… does that help us?*

Event Ordering: Distributed Systems

- 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 and Total Event Ordering

Define a notion of event ordering such that:

1. If A $\rightarrow$ 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.

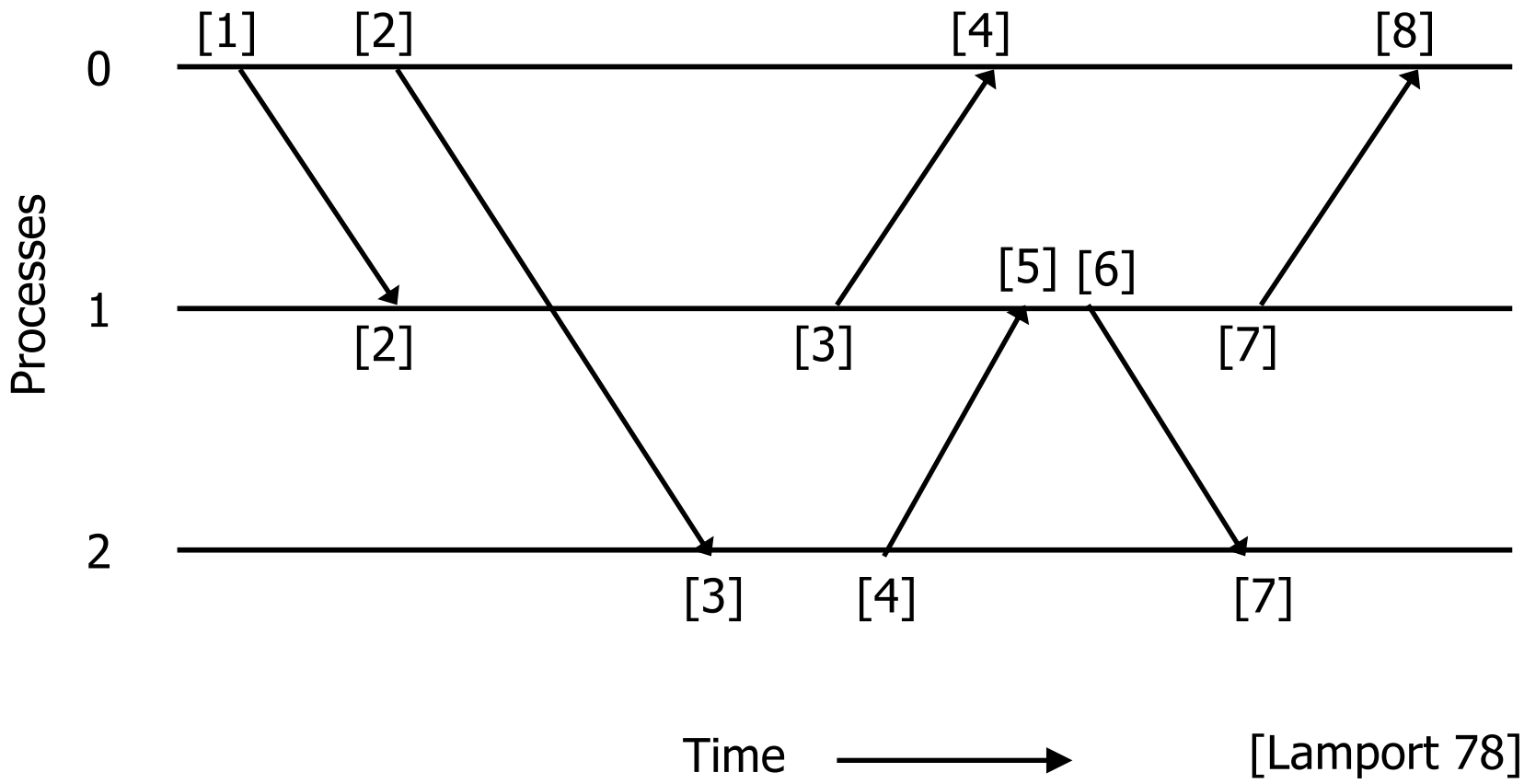
Solution:



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:

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

Logical Clocks: Example



Question

If $A \rightarrow B$ and $C \rightarrow B$, can we say $A \rightarrow C$?

A. Yes
B. No

Unique Challenges: Atomicity

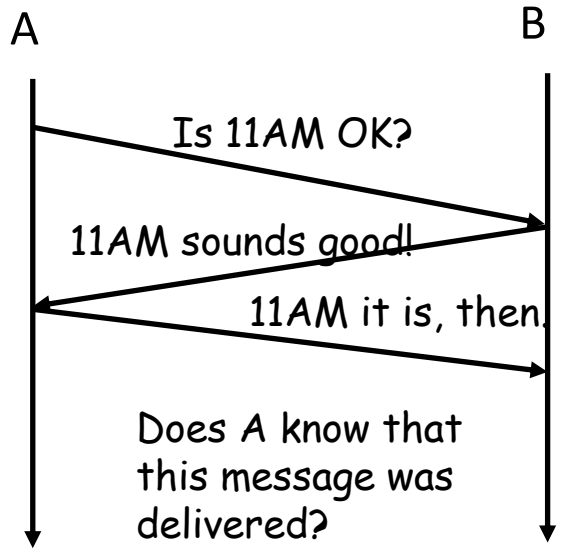
Atomicity in Distributed Systems

- How can we atomically update state on two different servers?
- 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?

Distributed Consensus: 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!



Question

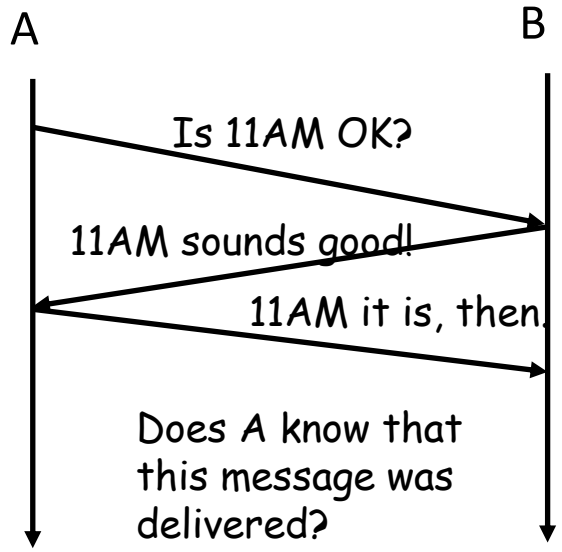
Distributed consensus in the presence of link failures is:

A. Possible
B. Not possible

Distributed Consensus: 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.

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.

Two-Phase Commit

Distributed Transactions

- Two or more 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
    - this may mean just writing it to a log
  - Write VOTE_COMMIT or VOTE_ABORT to their local logs
  - Send VOTE_COMMIT or VOTE_ABORT to coordinator

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

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 Queen has Died!
Long Live the Queen!

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

- 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 they 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 they have won.
  - If former alpha-process arises from dead, they bully 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.

Summary

- Parallel Computing: tightly-coupled systems
- Distributed Computing: loosely-coupled systems
- 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!)
  - elect a leader: many protocols, we looked at the bully algorithm