25. Distributed state/clock synchronization
Concurrency & Synchronization: Distributed Systems

Azer Bestavros
Computer Science Department
Boston University

Flynn Classification: SISD

Flynn Classification: SIMD

Flynn Classification: MIMD

Memory Topology: Shared

Memory Topology: Distributed
Memory Topology: Hybrid

Parallel vs. Distributed

- Parallel computing ~ a shared memory model
- Distributed computing ~ no shared memory

Inevitability of distributed computing

- Moore’s Law: "The density of transistors on a chip doubles every 18 months, for the same cost" (1965)
- Amendment: "... CPU clock speed cannot keep up" (2006)

Inevitability of distributed computing

- "Moore’s Law" constants as exponents matter! Processor-memory performance gap increases by ~ 50% per year...
- Shared memory cannot go beyond a fixed number of cores...

Distributed computing on a chip?!

... or data center
What does scale buy us?

- Rendering multiple frames of high-quality animation
- Simulating several hundred or thousand characters

But, how?

- It all boils down to...
  + Divide-and-conquer the problem
  + Throw more hardware at the problem
  + Develop abstractions to hide the mess below

Example: Pipelining

- Divide problem into stages
- Each stage consumes output from prior stage
- Each stage produces input to the next stage
- Make stages balanced to operate efficiently
- ... and make sure to synchronize them

Example: Divide & Conquer

<table>
<thead>
<tr>
<th>“Work”</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td>w2</td>
</tr>
<tr>
<td>worker1</td>
<td>worker2</td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
</tr>
</tbody>
</table>

| “Result” | Combine |

Choices, Choices, Choices

Assignment of workers to resources (scheduling!)
- Assign to different threads in the same core
- Assign to different in the same CPU
- Assign to different in a multi-processor system
- Assign to different in a distributed system

Choice of infrastructure
- Commodity vs. “exotic” hardware
- Number of machines, CPUs, cores per CPU
- Bandwidth of memory vs. disk vs. network
Challenges

- But this model is too simple!
- How do we assign work units to worker threads?
- What if we have more work units than threads?
- How do we aggregate the results at the end?
- How do we know all the workers have finished?
- What if the work cannot be divided into separate tasks?

For each of these problems multiple threads must communicate and/or access shared resources.

These are all distributed synchronization problems!

Distributed Synchronization

How is it different from what we have seen before?

- Entities are autonomous; nothing is really shared
  - No shared software (heterogeneous platforms, versions, …)
  - No shared resources (memory, file system, …)
  - No shared notion of time (no global clock)
- Communication via messaging
  - Asynchronous communication
  - Wires have state (e.g., packets in transit)
- Unreliability is inherent
  - Failures are to be expected
  - Messages may be lost
  - Malicious and adversarial (“Byzantine”) failures are possible

Example: Distributed Snapshot

- Problem: Can you record a consistent state of a distributed system?

- Motivation:
  - We need to have an accurate accounting of inventory managed in a distributed fashion.
  - We need to get accurate account balances for a distributed banking application.

Global State Consistency

Branch A

Branch B

Transfer
$100

@ 3:00pm Account has $100

Consistent

Account has $100

Global State Consistency

Branch A

Branch B

Transfer
$100

@ 3:00pm Account has $200

Inconsistent

Account has $100

Distributed Global Snapshot

- Assumptions (can be made realistic)
  - Communication is reliable; no messages are lost.
  - Message delivery delays can be arbitrary, but not infinite.
  - Messages are delivered in order.
  - There is no globally acceptable/consistent clock
  - All nodes must execute the same protocol
  - Any node can start the protocol at will
  - No node failures.
- How can we solve it?

Distributed Global Snapshot

- Key idea: Use a “marker” to flush link state into the receiver’s state.

Questions:
- How long would it take?
- Who starts the process?
- How do I collect the (distributed) answer?
- Can I take multiple snapshots concurrently?

Distributed Global Snapshot

- Upon receipt of 1st marker, record state and send markers on all outgoing links
- The state recorded at a node is
  - State when a first marker is received + changes resulting from messages received on link i, until marker is received on that link i
- Stop when markers are received on all incoming links

Answers:
- Can be initiated by multiple nodes at the same time.
- Terminates in finite time (assuming messages are delivered in finite time) proportional to the diameter of the graph.
- Collection of global state can be done through gossiping or by requiring initiating node to act as a “leader”.

Distributed Synchronization

- In global snapshot synchronization we assumed that there are no “global” clocks
  - Many synchronization problems can be addressed if we can synchronize clocks in a distributed system
- Let’s discuss “clock synchronization”…
Why clock synchronization matters?

Time keepers in history

- Astronomers: Look at the skies...
  - Oops... Earth rotation is slowing down

- Physicists: Look at atoms...
  - 9,192,631,770 cesium transitions == 1 solar second

- Computer Scientists: Look at CPU clock cycles...
  - 1 second is an eternity for a 4GHz CPU

Clock Synchronization Algorithms

- Centralized Algorithms
  - Cristian’s Algorithm (1989)
  - Berkeley Algorithm (1989)

- Distributed Algorithms
  - Averaging Algorithms (e.g., NTP)
  - Multiple External Time Sources

The Berkeley Algorithm (circa 1980s)

- The time daemon asks all the other machines for their clock values.
- The machines answer and the time daemon computes the average.
- The time daemon tells everyone how to adjust their clock.

Lets Go Back to Basics...

- Why do we need to agree on time? Could we live with something less constraining?
  - For most distributed algorithms/protocols, it is the consistency of the clocks that matters.

- Lamport: "what is important is that all processes agree on the order in which events occur."
  - Clocks do not have to be correct in the absolute sense, but only in the logical sense...
  - Say hello to "logical clocks" (or Lamport Clocks)

Event (Message) Ordering...

What’s wrong with the above picture?
Lamport Timestamps [1978]

The "Happens Before" Graph

- \( a \rightarrow b \) means that "a happens before b". It captures a weaker notion of causality.

"Happens Before" is observable in two situations:

1. If a and b are events in the same process, and a occurs before b, then \( a \rightarrow b \) is true.
2. If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then \( a \rightarrow b \) is also true.

Notes
- "Happens Before" is a transitive relation
- Two events are "concurrent" if neither happens before the other...

Finally a definition of concurrency

"Concurrency is absence of evidence to the contrary"

Example

- Identify the happens-before relations, by each rule of definition.
- Identify concurrent events.
- Is happens-before synonymous to causality?

Maintaining Lamport Clocks

- Requirement: If a and b are process events and a occurs before b, then \( a \rightarrow b \) is true.
  Enforcement: Each event has a timestamp (= local clock at time of event) and a process increment its logical clock counter for every event it encounters

- Requirement: If a is the event of a message transmittal and b is the event of the same message receipt, then \( a \rightarrow b \) is also true.
  Enforcement: Upon message receipt, advance local clock to exceed the timestamp of that message (if necessary)

- Requirement: Clocks are monotonically increasing; never ever go back in time...
  Enforcement: No need to enforce! The only operation on clocks will be to increment them (OK, will have to wrap around at some point...)

- Desirable: Never ever have two events with the same timestamp
  Enforcement: Tag the local clock with (unique) process ID. We have seen this before! Recall where?
Label the above events using Lamport timestamps.

So far:
- $a \rightarrow b$ implies that $TS(a) < TS(b)$

It may be desirable to have the converse implication correct as well, namely that
- $TS(a) < TS(b)$ implies that $a \rightarrow b$

We want a total order to be able to break ties (e.g., recall the Bakery algorithm)

Do you notice a problem?
How can we solve that?
- Idea: Make every reply a global snapshot problem
- Problem: Slow...
- Big Problem: What happens if some node dies?

In global snapshot synchronization we assumed that nodes do not fail and that messages are never lost...
- Distributed/cloud systems are of enormous scale; failure is the rule and not the exception!
- According to published reports from 2007 (10+ years ago) a single data center has over 1.3 million cores!

Let’s discuss “fault tolerance”...
Dealing with Failures

- Fault-tolerance through replication.
  - Need to ensure that replicas remain consistent.
  - Replicas must process requests in the same order.

Distributed Consensus

What we really want is for a set of servers to agree on a the state (of the world)

⇒ State Machine Replication

Simple solution:

⇒ A single node acts as the "decider"

What if the decider fails? Big problem!

Distributed Consensus

What we really want is for a set of servers to agree on a the state (of the world)

⇒ Where should we look for an answer?

Paxos!

Replicated State Machines

Devised by Leslie Lamport in 1989 as "The Part-Time Parliament" Problem

"Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxos parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems."

Rejected as unimportant and "confusing"!

Paxos: The Lost Manuscript

Published in 1998 as a "lost manuscript" in TOCS after it was put to use with a note from editor:

“This submission was recently discovered behind a filing cabinet in the TOCS editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.”

Today, the solution of this problem is what makes the cloud (and all our modern applications) work!
Fault-Tolerant Consensus

Requirements
Safety
- Only a value that has been proposed may be chosen.
- Only a single value is chosen.
- A process never learns that a value has been chosen until it actually has been.

Goals
Liveness
- Impossibility Proof (1985)

Assumptions
Failures
"Fail Stop" assumption
- When a node fails, it ceases to function entirely.
- May resume normal operation when restarted.

Messages
- May be lost.
- May be duplicated.
- May be delayed (and thus reordered).
- May not be corrupted.

Stable Storage

Paxos Roles (and Terms)

- **Proposer**
  - Suggests values for consideration by Acceptors.
  - Advocates for a client.

- **Acceptor**
  - Considers the values proposed by proposers.
  - Renders an accept/reject decision.

- **Learner**
  - Learns the chosen value.

In practice, each node will usually play all three roles.

Basic Paxos Algorithm

- **Phase 1a: Prepare**
  Select proposal number N and send a prepare(N) request to a quorum of acceptors.

- **Phase 1b: Promise**
  If N > number of any previous promises or acceptances,
  - promise to never accept any future proposal less than N,
  - send a promise(N, U) response (where U is the highest-numbered proposal accepted so far if any)

- **Phase 2a: Accept**
  If proposer received promise responses from a quorum,
  - send an accept(N, W) request to those acceptors (where W is the value of the highest-numbered proposal among the promise responses, or any value if no promise contained a proposal)

- **Phase 2b: Accepted**
  If N >= number of any previous promise,
  - accept the proposal
  - send an accepted notification to the learner

* = record to stable storage

It's a tough world out there!

- CAP Theorem states that you can choose only 2 of the following 3 properties:
  - C: Strong Consistency
  - A: 100% availability even if network is partitioned
  - P: tolerance of network partitions

- So, what are the choices?
  - CA is not realistic for cloud scale
  - AP is popular for noSQL (Apache CouchDB & AWS DynamoDB)
  - CP = Paxos (Google Spanner & FB HBase)

Lots of Consistency Models

- **Skiplist Consistency**
  Absolute time ordering of all shared accesses?
  No proposal has any shared access in the same order.
  Accesses are further more in order according to a consistent global timestamp.

- **Linearizability Consistency**
  All processes see identical accesses in the same order.
  Accesses are not ordered in time.
  Different processes may not always see the same order.

- **Sequential Consistency**
  All processes see identical accesses in the same order.
  Process interfere with each other in the order they occurred.
  All processes have different processes may not always see the same order.

- **Causal Consistency**
  All processes see identical accesses in the same order.
  Accesses are not ordered in time.
  Different processes may not always see the same order.

- **Weak Consistency**
  Shared data is not serialized to be consistent only after a consistent return.
  Data is not consistent when a consistent region is read.

- **Release Consistency**
  Shared data is not consistent when a consistent region on a consistent region is read.
  Consistent data is consistent when a consistent region is read.

- **Entry Consistency**
  Shared data is not consistent when a consistent region is read.
  Consistent data is consistent when a consistent region is read.

- **Two Phases**
  - Phase 1: Propose values
  - Phase 2: Accept values