DISTRIBUTED SYSTEMS (COMP9243)

Lecture 7 (A): Synchronisation and Coordination
Part 1

Slide 1

- Distributed Algorithms
- Time and Clocks
- Global State
- Concurrency Control

DISTRIBUTED ALGORITHMS

Algorithms that are intended to work in a distributed environment

- Used to accomplish tasks such as:
  - Communication
  - Accessing resources
  - Allocating resources
  - Consensus
  - etc.

Synchronisation and coordination inextricably linked to distributed algorithms

- Achieved using distributed algorithms
- Required by distributed algorithms

Synchronous vs Asynchronous Distributed Systems

Timing model of a distributed system

Slide 3

- Affected by:
  - Execution speed/time of processes
  - Communication delay
  - Clocks & clock drift

Synchronous Distributed System:

- Time variance is bounded

Execution: bounded execution speed and time

Communication: bounded transmission delay

Clocks: bounded clock drift (and differences in clocks)

Effect:

- Can rely on timeouts to detect failure
- Easier to design distributed algorithms
- Very restrictive requirements
  - Limit concurrent processes per processor Why?
  - Limit concurrent use of network Why?
  - Require precise clocks and synchronisation
Slide 5

Asynchronous Distributed System:
- Time variance is not bounded
  - Execution: different steps can have varying duration
  - Communication: transmission delays vary widely
  - Clocks: arbitrary clock drift

Effect:
- Allows no assumption about time intervals
- Cannot rely on timeouts to detect failure
- Most asynch DS problems hard to solve
- Solution for asynch DS is also a solution for synch DS
- Most real distributed systems are hybrid synch and asynch

Slide 6

EVALUATING DISTRIBUTED ALGORITHMS

Key Properties:
1. Safety: Nothing bad happens
2. Liveness: Something good eventually happens

General Properties:
- Performance
  - number of messages exchanged
  - response/wait time
  - delay, throughput: \(1/(\text{delay} + \text{execution time})\)
  - complexity: \(O()\)
- Efficiency
  - resource usage: memory, CPU, etc.
- Scalability
- Reliability
  - number of points of failure (low is good)

Slide 7

SYNCHRONISATION AND COORDINATION

Important:
- Doing the right thing at the right time.

Two fundamental issues:
- Coordination (the right thing)
- Synchronisation (the right time)

Slide 8

COORDINATION

Coordinate actions and agree on values.

Coordinate Actions:
- What actions will occur
- Who will perform actions

Agree on Values:
- Agree on global value
- Agree on environment
- Agree on state
SYNCHRONISATION

Ordering of all actions
- Total ordering of events
- Total ordering of instructions
- Total ordering of communication
- Ordering of access to resources
- Requires some concept of time

MAIn issues

Time and Clocks: synchronising clocks and using time in distributed algorithms
Global State: how to acquire knowledge of the system’s global state
Concurrency Control: coordinating concurrent access to resources

TIME AND CLOCKS

TIME

Global Time:
- ‘Absolute’ time
  - Einstein says no absolute time
  - Absolute enough for our purposes
- Astronomical time
  - Based on earth’s rotation
  - Not stable
- International Atomic Time (IAT)
  - Based on oscillations of Cesium-133
- Coordinated Universal Time (UTC)
  - Leap seconds
  - Signals broadcast over the world
Local Time:

- Relative not ‘absolute’
- Not synchronised to Global source

---

**Physical Clocks**

Based on actual time:

- $C_p(t)$: current time (at UTC time $t$) on machine $p$
- Ideally $C_p(t) = t$
- $x$ Clock differences causes clocks to drift
- Must regularly synchronise with UTC

**Computer Clocks:**

- Crystal oscillates at known frequency
- Oscillations cause timer interrupts
- Timer interrupts update clock

**Clock Skew:**

- Crystals in different computers run at slightly different rates
- Clocks get out of sync
- Skew: instantaneous difference
- Drift: rate of change of skew

---

**Using Clocks in Computers**

**Timestamps:**

- Used to denote at which time an event occurred

**Synchronisation Using Clocks:**

- Performing events at an exact time (turn lights on/off, lock/unlock gates)
- Logging of events (for security, for profiling, for debugging)
- Tracking (tracking a moving object with separate cameras)
- Make (edit on one computer build on another)
- Ordering messages
**Synchronising Physical Clocks**

**Internal Synchronisation:**
- Clocks synchronise locally
- Only synchronised with each other

**External Synchronisation:**
- Clocks synchronise to an external time source
- Synchronise with UTC every $\delta$ seconds

**Time Server:**
- Server that has the correct time
- Server that calculates the correct time

---

**Cristian’s Algorithm**

**Time Server:**
- Has UTC receiver
- Passive

**Algorithm:**
- Clients periodically request the time
- Don’t set time backward Why not?
- Take propagation and interrupt handling delay into account
  - $(T1 - T0)/2$
  - Or take a series of measurements and average the delay
- Accuracy: 1-10 millisecond (RTT in LAN)

*What is a drawback of this approach?*

---

**Berkeley Algorithm**

*Accuracy: 20-25 milliseconds*

*When is this useful?*

---

**Network Time Protocol (NTP)**

**Hierarchy of Servers:**
- Primary Server: has UTC clock
- Secondary Server: connected to primary
- etc.

**Synchronisation Modes:**

**Multicast:** for LAN, low accuracy

**Procedure Call:** clients poll, reasonable accuracy

**Symmetric:** Between peer servers, highest accuracy
Synchronisation:
- Estimate clock offsets and transmission delays between two nodes
- Keep estimates for past communication
- Choose offset estimate for lowest transmission delay
- Also determine unreliable servers
- Accuracy 1 - 50 msec

Logical Clocks

Event ordering is more important than physical time:
- Events (e.g., state changes) in a single process are ordered
- Processes need to agree on ordering of causally related events (e.g., message send and receive)

Local ordering:
- System consists of \( N \) processes \( p_i, i \in \{1, \ldots, N\} \)
- Local event ordering \( \rightarrow \)
  - If \( p_i \) observes \( e \) before \( e' \), we have \( e \rightarrow e' \)

Global ordering:
- Leslie Lamport’s happened before relation \( \rightarrow \)
  - Smallest relation, such that
    1. \( e \rightarrow e' \) implies \( e 
      \rightarrow e'' \)
    2. For every message \( m \), \( \text{send}(m) \rightarrow \text{receive}(m) \)
    3. Transitivity: \( e \rightarrow e' \) and \( e' \rightarrow e'' \) implies \( e \rightarrow e'' \)

The relation \( \rightarrow \) is a partial order:
- If \( a \rightarrow b \), then \( a \) causally affects \( b \)
- We consider unordered events to be concurrent:

Example:
\[ E_{11} \rightarrow E_{12} \rightarrow E_{13} \rightarrow E_{14} \rightarrow p_1 \]
\[ E_{21} \rightarrow E_{22} \rightarrow E_{23} \rightarrow E_{24} \rightarrow p_1 \]

- Causally related: \( E_{11} \rightarrow E_{12}, E_{13}, E_{14}, E_{21} \rightarrow E_{24}, \ldots \)
- Concurrent: \( E_{11} || E_{21}, E_{12} || E_{22}, E_{13} || E_{23}, E_{14} || E_{24}, \ldots \)
Locks and Vector Clocks

**Lamport’s logical clocks:**
- Software counter to locally compute the happened-before relation →
- Each process \( p_i \) maintains a logical clock \( L_i \)

**Lamport timestamp:**
- \( L_i(e) \): timestamp of event \( e \) at \( p_i \)
- \( L(e) \): timestamp of event \( e \) at process it occurred at

**Implementation:**
1. Before timestamping a local event \( p_i \), executes \( L_i \) := \( L_i + 1 \)
2. Whenever a message \( m \) is sent from \( p_i \) to \( p_j \):
   - \( p_j \) executes \( L_j := L_i + 1 \) and sends \( L_j \) with \( m \)
   - \( p_j \) receives \( L_j \) with \( m \) and executes

**Properties:**
- \( a \rightarrow b \) implies \( L(a) < L(b) \)
- \( L(a) < L(b) \) does not necessarily imply \( a \rightarrow b \)

**Example:**

![Diagram of Lamport’s logical clocks example]

**Total event ordering:**
- Complete partial to total order by including process identifiers
- Given local time stamps \( L_i(e) \) and \( L_i(e') \), we define global time stamps \((L_i(e), i)\) and \((L_j(e'), j)\)

**Lexicographical ordering:** \( (L_i(e), i) < (L_j(e'), j) \) if
  - \( L_i(e) < L_j(e') \) or
  - \( L_i(e) = L_j(e') \) and \( i < j \)

**Implementation:**

**Vector Clocks**

**Main shortcoming of Lamport’s clocks:**
- \( L(a) < L(b) \) does not imply \( a \rightarrow b \)
- We cannot deduce causal dependencies from time stamps:

![Diagram of Vector Clocks example]

- We have \( L_1(E_{11}) < L_3(E_{33}) \), but \( E_{11} \not\rightarrow E_{33} \)
- Why?
  - Clocks advance independently or via messages
  - There is no history as to where advances come from
Vector clocks:
- At each process, maintain a clock for every other process
- i.e., each clock $V_i$ is a vector of size $N$
- $V[j]$ contains $i$’s knowledge about $j$’s clock
- Events are timestamped with a vector

Implementation:
1. Initially, $V_{ij} := 0$ for $i, j \in \{1, \ldots, N\}$
2. Before $p_i$ timestamps an event: $V_{ij} := V_{ij} + 1$
3. Whenever a message $m$ is sent from $p_i$ to $p_j$:
   - $p_i$ executes $V[i] := V[i] + 1$ and sends $V_i$ with $m$
   - $p_j$ receives $V_i$ with $m$ and merges the vector clocks $V_i$ and $V_j$

\[
V_j[k] = \begin{cases} 
\max(V_j[k], V_i[k]) + 1 & \text{if } j = k \\
\max(V_j[k], V_i[k]) & \text{otherwise}
\end{cases}
\]

Properties:
- For all $i, j, V_i[j] \geq V_j[j]$
- $a \rightarrow b$ if $V(a) < V(b)$ where
  - $V = V''$ if $V_i = V''$ for $i \in \{1, \ldots, N\}$
  - $V \geq V''$ if $V_i \geq V''$ for $i \in \{1, \ldots, N\}$
  - $V > V''$ if $V_i > V''$ for $i \in \{1, \ldots, N\}$
  - $V\parallel V'$ if $V \geq V' \land V' \geq V$

Example:
- For $L_1(E_{12})$ and $L_2(E_{12})$, $2 = 2$ versus $(2, 0, 0) \neq (0, 0, 2)$

Homework
- How would you use vector clocks to implement causal consistency?
- Could you use logical clocks to implement sequential consistency?

Reading List
Optional

Slide 29

Time, Clocks, and the Ordering of Events in a Distributed system
Classic on Lamport clocks.

Distributed Snapshots: Determining Global States of Distributed Systems
Chandy and Lamport algorithm.