DISTRIBUTED SYSTEMS (COMP9243)

Lecture 7 (A): Synchronisation and Coordination
Part 1

Slide 1

1. Distributed Algorithms
2. Time and Clocks
3. Global State
4. 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
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 async DS problems hard to solve
- Solution for async DS is also a solution for synch DS
- Most real distributed systems are hybrid synch and async

---

**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)

---

**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 and Coordination**

Important:

- Doing the right thing at the right time.

Two fundamental issues:
- Coordination (the right thing)
- Synchronisation (the right time)
**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
### Physical Clocks

**Based on actual time:**

- \( C_p(t) \): current time (at UTC time \( t \)) on machine \( p \)
- Ideally \( C_p(t) = t \)
- Clock differences cause 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

---

**Local Time:**

- Relative not ‘absolute’
- Not synchronised to Global source
Accuracy: 20-25 milliseconds
When is this useful?

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
  - \((T_1 - T_0)/2\)
  - Or take a series of measurements and average the delay
- Accuracy: 1-10 millsec (RTT in LAN)

What is a drawback of this approach?
LAMPORT

Safety, Liveness
Logical clocks and vector clocks
Snapshots
Byzantine generals
Paxos consensus
TLA+, LaTeX
Turing Award 2013

Comments about his papers: Google: lamport my writings

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 $\rightarrow_i$:
  - If $p_i$ observes $e$ before $e'$, we have $e \rightarrow_i e'$

Global ordering:
- Leslie Lamport’s happened before relation $\rightarrow$
- Smallest relation, such that
  1. $e \rightarrow_i e'$ 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 $\not\rightarrow a$ and $b \not\rightarrow a$ implies $a \parallel b$

Lamport’s logical clocks:
- Software counter to locally compute the happened-before relation $\rightarrow$
- Each process $p_i$ maintains a logical clock $L_i$
- Implementation:
  1. Before timestamping a local event $e$, $p_i$ executes $L_i := L_i + 1$
  2. Whenever a message $m$ is sent from $p_i$ to $p_j$:
    - $p_i$ executes $L_i := L_i + 1$ and sends $L_i$ with $m$
    - $p_j$ receives $L_i$ with $m$ and executes $L_j := \max(L_j, L_i) + 1$
    (receive($m$) is annotated with the new $L_j$)

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

Example:

Slide 25

How can we order $E_{13}$ and $E_{23}$?

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 $(L_i(e'), j)$

Slide 26

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

$E_{13} = 3$, $E_{24} = 4$. Did $E_{13}$ happen before $E_{24}$?

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:

Slide 27

- We have $L_i(E_{11}) < L_j(E_{33})$, 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 $N$
- $V_i[j]$ contains $i$’s knowledge about $j$’s clock
- Events are timestamped with a vector

Implementation:

1. Initially, $V_i[j] := 0$ for $i, j \in \{1, \ldots, N\}$
2. Before $p_k$ timestamps an event: $V_i[i] := V_i[i] + 1$
3. Whenever a message $m$ is sent from $p_i$ to $p_j$:
   - $p_j$ executes $V_j[i] := V_j[i] + 1$ and sends $V_i$ with $m$
   - $p_j$ receives $V_i$ with $m$ and merges the vector clocks $V_i$ and $V_j$:
     
     $V_j[i] := \begin{cases} 
     \max(V_j[k], V_i[k]) + 1, & \text{if } j = k \\
     \max(V_j[k], V_i[k]), & \text{otherwise}
     \end{cases}$
Properties:

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

Example:

<table>
<thead>
<tr>
<th>State</th>
<th>Time</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1_1</td>
<td>1.0,0</td>
<td>P1</td>
</tr>
<tr>
<td>E1_2</td>
<td>2.0,0</td>
<td></td>
</tr>
<tr>
<td>E1_4</td>
<td>2.2,0</td>
<td></td>
</tr>
<tr>
<td>E1_5</td>
<td>2.3,1</td>
<td></td>
</tr>
<tr>
<td>E1_6</td>
<td>2.4,1</td>
<td></td>
</tr>
<tr>
<td>E3_1</td>
<td>0.0,1</td>
<td></td>
</tr>
<tr>
<td>E3_2</td>
<td>0.0,2</td>
<td></td>
</tr>
<tr>
<td>E3_3</td>
<td>0.0,1</td>
<td></td>
</tr>
</tbody>
</table>

- For $L_1(E_{13})$ and $L_2(E_{12})$, $2 = 2$ versus $(2, 0, 0) \neq (0, 0, 2)$

Global State

Determining global properties:

- Distributed garbage collection: Do any references exist to a given object?
- Distributed deadlock detection: Do processes wait in a cycle for each other?
- Distributed termination detection: Did a set of processes cease all activity? (Consider messages in transit?)
- Distributed checkpoint: What is a correct state of the system to save?

Consistent Cuts

Determining global properties:

- We need to combine information from multiple nodes
- Without global time, how do we know whether collected local information is consistent?
- Local state sampled at arbitrary points in time surely is not consistent
- We need a criterion for what constitutes a globally consistent collection of local information
Local history:
- \( N \) processes \( p_i, i \in \{1, \ldots, N\} \)
- For each \( p_i \):
  - event: \( e_i \) local action or communication
  - history: \( h^i_k = (e^i_0, e^i_1, \ldots, e^i_k) \)
  - May be finite or infinite

Process state:
- \( s^i_k \): state of process \( p_i \) immediately before event \( e^i_k \)
- \( s^i_k \) records all events included in the history \( h^i_{k-1} \)
- Hence, \( s^i_0 \) refers to \( p_i \)'s initial state

Global history and state:
- Using a total event ordering, we can merge all local histories into a global history:
  \[ H = \bigcup_{i=1}^{N} h_i \]
- Similarly, we can combine a set of local states \( s_1, \ldots, s_N \) into a global state:
  \[ S = (s_1, \ldots, s_N) \]
- Which combination of local state is consistent?

Cuts:
- Similar to the global history, we can define cuts based on \( k \)-prefixes:
  \[ C = \bigcup_{i=1}^{N} h^i_k \]
- \( h^i_k \) is history of \( p_i \) up to and including event \( e^i_k \)
- The cut \( C \) corresponds to the state
  \[ S = (s^{i+1}_1, \ldots, s^{i+1}_N) \]
- The final events in a cut are its frontier:
  \[ \{ e^i_k \mid i \in \{1, \ldots, N\} \} \]
Consistent cut:

- We call a cut **consistent** if, for all events $e' \in C, e \rightarrow e'$ implies $e \in C$.
- A global state is consistent if it corresponds to a consistent cut.
- Note: we can characterise the execution of a system as a sequence of consistent global states $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots$.

Linearisation:

- A global history that is consistent with the happened-before relation $\rightarrow$ is also called a **linearisation** or **consistent run**.
- A linearisation only passes through consistent global states.
- A state $S'$ is **reachable** from state $S$ if there is a linearisation that passes through $S$ and then $S'$.

**Chandy & Lamport’s Snapshots**

- Determines a consistent global state.
- Takes care of messages that are in transit.
- Useful for evaluating stable global properties.

Properties:

- Reliable communication and failure-free processes.
- Point-to-point message delivery is ordered.
- Process/channel graph must be strongly connected.
- On termination, processes hold only their local state components and a set of messages that were in transit during the snapshot.

Outline of the algorithm:

1. One process initiates the algorithm by:
   - recording its local state and
   - sending a **marker message** $\ast$ over each outgoing channel.
2. On receipt of a marker message over incoming channel $c$,
   - if local state not yet saved, save local state and send marker messages, or
   - if local state already saved, channel snapshot for $c$ is complete.
3. Local contribution complete after markers received on all incoming channels.

Result for each process:

- One local state snapshot.
- For each incoming channel, a set of messages received after performing the local snapshot and before the marker came down that channel.
**Spanner and TrueTime**

Globally Distributed Database
- Want external consistency (linearisability)
- Want lock-free read transactions (for scalability)

WWGD? (what would Google do?)

**External Consistency with a Global Clock**

Data:
- versioned using timestamp

Read:
- Read operations performed on a snapshot
- Snapshot: latest version of data items <= given timestamp

Write:
- Each write operation (transaction actually) has unique timestamp
  - Timestamps must not overlap!
- Write operations are protected by locks
  - Means they don’t overlap
- So get global time during the transaction
  - Means timestamps won’t overlap

**But clocks are not perfectly synchronised.**

So transaction A could get the same timestamp as transaction B
TRUE TIME

Add uncertainty to timestamps:
- \( \text{now}() \): current local clock value
- \( \text{now}().\text{earliest}, \text{now}().\text{latest} \): maximum skew of clock

Add delay to transaction:
- so timestamps can't possibly overlap
- \( s = \text{now}(); \) wait until \( \text{now}().\text{earliest} > s.\text{latest} \)

SYNCHRONISATION

TrueTime Architecture

Client

Datacenter 1  Datacenter 2  \ldots  Datacenter n

Compute reference [earliest, latest] = \( \text{now} \pm \epsilon \)
Concurrency

Concurrency in a Non-Distributed System:

Typical OS and multithreaded programming problems

- Prevent race conditions
- Critical sections
- Mutual exclusion
  - Locks
  - Semaphores
  - Monitors
- Must apply mechanisms correctly
  - Deadlock
  - Starvation

Concurrency in a Distributed System:

Distributed System introduces more challenges

- No directly shared resources (e.g., memory)
- No global state
- No global clock
- No centralised algorithms
- More concurrency

Distributed Mutual Exclusion

- Concurrent access to distributed resources
- Must prevent race conditions during critical regions

Requirements:

- Safety: At most one process may execute the critical section at a time
- Liveness: Requests to enter and exit the critical section eventually succeed
- Ordering: Requests are processed in happened-before ordering (also Fairness)

Recall: Evaluating Distributed Algorithms

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)
**METHOD 1: CENTRAL SERVER**

Simplest approach:
- Requests to enter and exit a critical section are sent to a lock server
- Permission to enter is granted by receiving a token
- When critical section left, token is returned to the server

![Diagram of method 1](image)

Properties:
- Number of message exchanged?
- Delay before entering critical section?
- Reliability?
- Easy to implement
- Does not scale well
- Central server may fail

---

**METHOD 2: TOKEN RING**

Implementation:
- All processes are organised in a logical ring structure
- A token message is forwarded along the ring
- Before entering the critical section, a process has to wait until the token comes by
- Must retain the token until the critical section is left

![Diagram of method 2](image)

Properties:
- Number of message exchanged?
- Delay before entering critical section?
- Reliability?
- Ring imposes an average delay of $N/2$ hops (limits scalability)
- Token messages consume bandwidth
- Failing nodes or channels can break the ring (token might be lost)

---

**METHOD 3: USING MULTICASTS AND LOGICAL CLOCKS**
**Method 3: Using Multicasts and Logical Clocks**

Algorithm by Ricart & Agrawala:
- Processes \( p_i \) maintain a Lamport clock and can communicate pairwise
- Processes are in one of three states:
  1. **Released**: Outside of critical section
  2. **Wanted**: Waiting to enter critical section
  3. **Held**: Inside critical section

**Process behaviour:**
1. If a process wants to enter, it
   - multicasts a message \((L_i, p_i)\) and
   - waits until it has received a reply from every process
2. If a process is in **Released**, it immediately replies to any request to enter the critical section
3. If a process is in **Held**, it delays replying until it is finished with the critical section
4. If a process is in **Wanted**, it replies to a request immediately only if the requesting timestamp is **smaller** than the one in its own request

**Properties:**
- Number of message exchanged?
- Delay before entering critical section?
- Reliability?
- Multicast leads to increasing overhead (try using only subsets of peer processes)
- Susceptible to faults

**Mutual Exclusion: A Comparison**

**Messages Exchanged:**
- Messages per entry/exit of critical section
  - Centralised: 3
  - Ring: 1 → ∞
  - Multicast: \(2(n - 1)\)

**Delay:**
- Delay before entering critical section
  - Centralised: 2
  - Ring: 0 → n - 1
  - Multicast: \(2(n - 1)\)

**Reliability:**
- Problems that may occur
  - Centralised: coordinator crashes
  - Ring: lost token, process crashes
  - Multicast: any process crashes
HOME WORK

Slide 61

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

Slide 62

Hacker’s edition:

→ Modify the Ricart Agrawala mutual exclusion algorithm to only require sending to a subset of the processes.
→ Can you modify the centralised mutual exclusion algorithm to tolerate coordinator crashes?

Slide 63

READING LIST

Optional

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.