DISTRIBUTED SYSTEMS (COMP9243)

Lecture 4: Synchronisation and Coordination

(Part 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

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 \textit{Why}?
- Limit concurrent use of network \textit{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 sync 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: \( \frac{1}{(\text{delay} + \text{execution time})} \)
  • complexity: \( O() \)

➜ Efficiency
  • resource usage: memory, CPU, etc.

➜ Scalability

➜ Reliability
  • number of points of failure (low is good)
SYNCHRONISATION AND COORDINATION

Important:

Doing the right thing at the right time.

Two fundamental issues:

→ Coordination (the right thing)
→ Synchronisation (the right time)
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
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
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
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 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
SYNCHRONIZE ALL THE CLOCKS

PHYSICAL CLOCKS
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
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
  - \( \frac{(T_1 - T_0)}{2} \)
  - Or take a series of measurements and average the delay
- Accuracy: 1-10 millisec (RTT in LAN)

What is a drawback of this approach?
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
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
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 $m$, $send(m) \rightarrow receive(m)$
  3. Transitivity: $e \rightarrow e'$ and $e' \rightarrow e''$ implies $e \rightarrow e''$.
The relation $\rightarrow$ is a partial order:

$\rightarrow$ If $a \rightarrow b$, then $a$ causally affects $b$

$\rightarrow$ We consider unordered events to be concurrent:

Example: $a \not\rightarrow b$ and $b \not\rightarrow a$ implies $a \parallel b$

$E_{11} \rightarrow E_{12}, E_{13}, E_{14}, E_{23}, E_{24}, \ldots$

$E_{21} \rightarrow E_{22}, E_{23}, E_{24}, E_{13}, E_{14}, \ldots$

$E_{11} \parallel E_{21}, E_{12} \parallel E_{22}, E_{13} \parallel E_{23}, E_{11} \parallel E_{22}, E_{13} \parallel E_{24}, E_{14} \parallel E_{23}, \ldots$

Logical 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 $p_j$:
  • $p_i$ 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 $a \rightarrow b$
Example:

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 $L_j(e')$, we define global time stamps $\langle L_i(e), i \rangle$ and $\langle L_j(e'), j \rangle$
- Lexicographical ordering: $\langle L_i(e), i \rangle < \langle L_j(e'), j \rangle$ iff
  - $L_i(e) < L_j(e')$ or
  - $L_i(e) = L_j(e')$ 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:

- 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 $N$
- $V_i[j]$ 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 $p_i$ timestamps an event: $V_i[i] := V_i[i] + 1$
③ Whenever a message $m$ is sent from $p_i$ to $p_j$:
   • $p_i$ executes $V_i[i] := V_i[i] + 1$ 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 $i, j$, $V_i[i] \geq V_j[i]$
- $a \rightarrow b$ 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 $V \nless V' \land V' \nless V$

Example:

- For $L_1(E_{12})$ and $L_3(E_{32})$, $2 = 2$ versus $(2, 0, 0) \neq (0, 0, 2)$
 GLOBAL STATE
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:

$\rightarrow N$ processes $p_i, i \in \{1, \ldots, N\}$

$\rightarrow$ For each $p_i$,
- event: $e^j_i$ local action or communication
- history: $h^k_i = \langle e^0_i, e^1_i, \ldots e^k_i \rangle$
- May be finite or infinite

Process state:

$\rightarrow s^k_i$ : state of process $p_i$ immediately before event $e^k_i$

$\rightarrow s^k_i$ records all events included in the history $h^{k-1}_i$

$\rightarrow$ Hence, $s^0_i$ 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:

$$C = \bigcup_{i=1}^{N} h_i^c$$

- $h_i^c$ is history of $p_i$ up to and including event $e_i^c$

- The cut $C$ corresponds to the state

$$S = (s_1^{c_1+1}, \ldots, s_N^{c_N+1})$$

- The final events in a cut are its frontier:

$$\{e_i^c \mid i \in \{1, \ldots, N\}\}$$
Consistent cut:

→ We call a cut consistent iff,

\[ \text{for all events } e' \in C, e \rightarrow e' \implies e \in C \]

→ 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 thorough \( S \) 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 * 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 TRUE TIME

Globally Distributed Database

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

WWGD? (what would Google do?)
USE A GLOBAL CLOCK!
EXTERNAL CONSISTENCY WITH A GLOBAL CLOCK

Data:
- versioned using timestamp

Read:
- Read operations performed on a snapshot
- Snapshot: latest version of data items $\leq$ 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:

- `TT.now()`: current local clock value
- `TT.now().earliest(), TT.now().latest`: maximum skew of clock

Add delay to transaction:

- so timestamps can’t possibly overlap
- `s = TT.now(); wait until TT.now().earliest > s.latest`
TRUE TIME ARCHITECTURE

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

SYNCHRONISATION

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:**

1. **Safety:** At most one process may execute the critical section at a time
2. **Liveness:** Requests to enter and exit the critical section eventually succeed
3. **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.

\[\begin{array}{c}
\text{Coordinator} \\
\text{Queue is empty}
\end{array}\]
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
Properties:

- Number of messages 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

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 \( \langle L_i, p_i \rangle \) 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 messages exchanged?
- Delay before entering the critical section?
- Reliability?
- Multicast leads to increasing overhead (try using only subsets of peer processes)
- Susceptible to faults

**Method 3: Using Multicasts and Logical Clocks**
Mutual Exclusion: A Comparison

Messages Exchanged:

- Messages per entry/exit of critical section
  - Centralised: 3
  - Ring: $1 \rightarrow \infty$
  - Multicast: $2(n - 1)$

Delay:

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

Reliability:

- Problems that may occur
  - Centralised: coordinator crashes
  - Ring: lost token, process crashes
  - Multicast: any process crashes
HOMEWORK

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

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?
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.