
DISTRIBUTED SYSTEMS (COMP9243)

Lecture 5: 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.

Slide 2

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
- (Partial) failure

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)

Slide 4

Effect:

- Can rely on timeouts to detect failure
 - ✓ Easier to design distributed algorithms
 - ✗ Very restrictive requirements
 - Limit concurrent processes per processor
 - Limit concurrent use of network
 - 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

Slide 5 **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

EVALUATING DISTRIBUTED ALGORITHMS

General Properties:

- Slide 6**
- Performance
 - number of messages exchanged
 - response/wait time
 - delay
 - throughput: $1/(delay + executiontime)$
 - complexity: $O()$
 - Efficiency
 - resource usage: memory, CPU, etc.
 - Scalability
 - Reliability
 - number of points of failure (low is good)

SYNCHRONISATION AND COORDINATION

Important:

Slide 7 Doing the right thing at the right time.

Two fundamental issues:

- Coordination (the right thing)
- Synchronisation (the right time)

SYNCHRONISATION

Ordering of all actions

- Slide 8**
- Total ordering of events
 - Total ordering of instructions
 - Total ordering of communication
 - Ordering of access to resources
 - Requires some concept of time

COORDINATION

Coordinate actions and agree on values.

Coordinate Actions:

- What actions will occur
- Who will perform actions

Slide 9

Agree on Values:

- Agree on global value
 - Agree on environment
 - Agree on state
-
-

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

Coordination: when do processes need to coordinate and how do they do it

TIME AND CLOCKS

Slide 11

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:

- Slide 13**
- Not synchronised to Global source
 - Relative not 'absolute'



CLOCKS

Computer Clocks:

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

Slide 14

Clock Skew:

- Crystals in different computers run at slightly different rates
- Clocks get out of sync
- Maximum drift rate

Timestamps:

- Used to denote at which time an event occurred
- Logical vs Physical



PHYSICAL CLOCKS

Synchronisation Using Physical Clocks:

Examples:

- Slide 15**
- 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)



Based on actual time

- Slide 16**
- $C_p(t)$: current time (at UTC time t) on machine p
 - Ideally $C_p(t) = t$
 - ✗ Clock skew causes clocks to *drift*
 - Must regularly synchronise with UTC



SYNCHRONISING PHYSICAL CLOCKS

External Synchronisation:

- Clocks synchronise to an external time source
- Synchronise with UTC every δ seconds

Slide 17

Internal Synchronisation:

- Clocks synchronise locally
- Only synchronised with each other

Time Server:

- Server that has the correct time
- Server that calculates the correct time

CRISTIAN'S ALGORITHM

Time Server:

- Has UTC receiver
- Passive

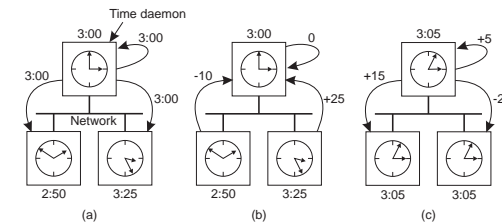
Slide 18

Algorithm:

- Clients periodically request the time
- Don't set time backward
- Take propagation and interrupt handling delay into account
 - $(T1 - T0)/2$
 - Or take a series of measurements and average the delay

BERKELEY ALGORITHM

Slide 19



NETWORK TIME PROTOCOL (NTP)

Hierarchy of Servers:

- Primary Server: has UTC clock
- Secondary Server: connected to primary
- etc.

Slide 20

Synchronisation Modes:

Multicast: for LAN, low accuracy

Procedure Call: clients poll, reasonable accuracy

Symmetric: Between peer servers. highest accuracy

Slide 21

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 $p_i, i \in \{1, \dots, N\}$

Slide 22

- Local event ordering \rightarrow_i :

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

Global ordering:

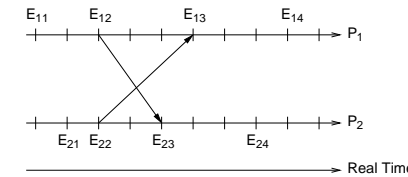
- Leslie Lamport's happened before relation \rightarrow
- Smallest relation, such that
 1. $e \rightarrow_i e'$ implies $e \rightarrow e'$
 2. For every message m , $send(m) \rightarrow receive(m)$
 3. Transitivity: $e \rightarrow e'$ and $e' \rightarrow e''$ implies $e \rightarrow e''$

The relation \rightarrow is a partial order:

- If $a \rightarrow b$, then a causally affects b
- We consider unordered events to be concurrent:

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

Slide 23



- Causally related: $E_{11} \rightarrow E_{12}, E_{13}, E_{14}, E_{23}, E_{24}, \dots$
 $E_{21} \rightarrow E_{22}, E_{23}, E_{24}, E_{13}, E_{14}, \dots$
- Concurrent: $E_{11} \parallel E_{21}, E_{12} \parallel E_{22}, E_{13} \parallel E_{23}, E_{11} \parallel E_{22}, E_{13} \parallel E_{24}, E_{14} \parallel E_{23}, \dots$

Lamport's logical clocks:

- Software counter to locally compute the happened-before relation \rightarrow
- 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

Slide 24

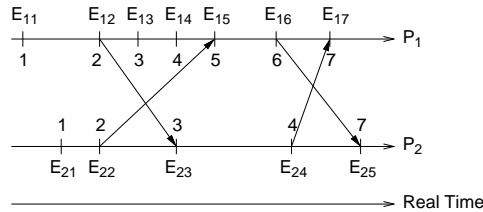
Implementation:

- 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:



Slide 25

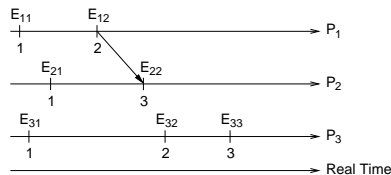
Total event ordering:

- 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 $i < j$

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 26

- We have $L_1(E_{11}) < L_3(E_{33})$, but $E_{11} \not\rightarrow E_{33}$
- Why?
 - Clocks advances 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

Implementation:

- ① Initially, $V_i[j] := 0$ for $i, j \in \{1, \dots, N\}$
- 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 :

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

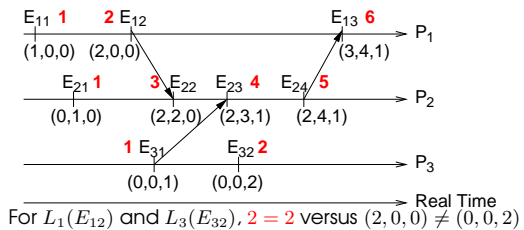
Slide 27

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 $i \in \{1, \dots, N\}$
 - $V \geq V'$ iff $V[i] \geq V'[i]$ for $i \in \{1, \dots, N\}$
 - $V > V'$ iff $V \geq V' \wedge V \neq V'$
 - $V \parallel V'$ iff $V \not\geq V' \wedge V' \not\geq V$

Example:

Slide 28



Slide 29

GLOBAL STATE

Slide 30

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!)

All these properties are **stable**: once they occur, they do not disappear without outside intervention.

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

Slide 31

Local history:

- N processes $p_i, i \in \{1, \dots, N\}$
- For each p_i ,
 - **event series** $\langle e_i^0, e_i^1, e_i^2, \dots \rangle$
 - is called p_i 's *history* denoted by h_i .
 - May be finite or infinite
- We denote by h_i^k a k -*prefix* of h_i .
- Each event e_i^k is either a local event or a communication event

Slide 32

Process state:

- State of process p_i immediately before event e_i^k denoted s_i^k
- State 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**:

$$H = \bigcup_{i=1}^N h_i$$

Slide 33

- Similarly, we can combine a set of local states s_1, \dots, s_N into a global state:

$$S = (s_1, \dots, s_N)$$

- Which combination of local state is **consistent**?

Cuts:

- Similar to the global history, we can define **cuts** based on k -prefixes:

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

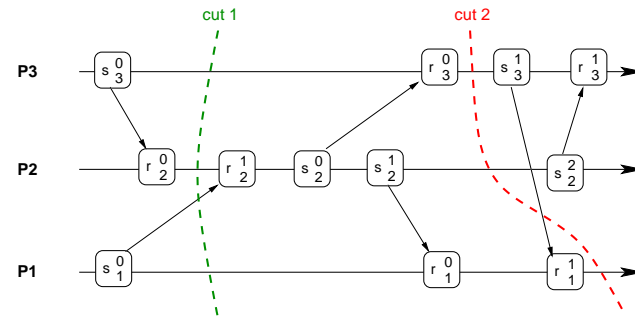
Slide 34

- $h_i^{c_i}$ is history of p_i up to and including event $e_i^{c_i}$
- The cut C **corresponds** to the state

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

- The final events in a cut are its **frontier**:

$$\{e_i^{c_i} \mid i \in \{1, \dots, N\}\}$$



Slide 35

Consistent cut:

- We call a cut **consistent** iff,

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

- 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

Slide 36

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots$$

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:

Slide 37

- 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:

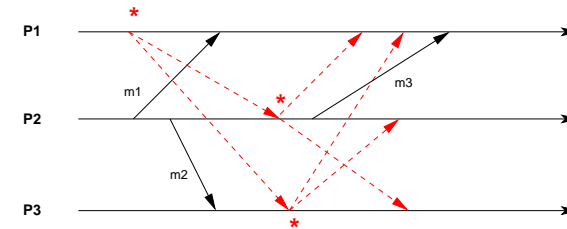
- ① One process initiates the algorithm by
 - recording its local state and
 - sending a **marker message** over each outgoing channel
- ② On receipt of a marker message over incoming channel c ,
 - if local state not yet saved, save local state and send marker messages, or
 - if local state already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

Slide 38

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

Slide 39



Slide 40

CONCURRENCY

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

Slide 41

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

Slide 42

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

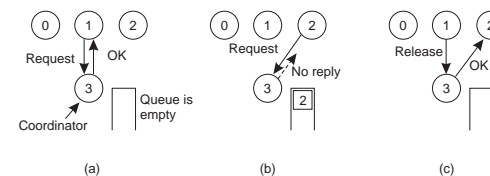
Slide 43

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

Slide 44



Slide 45

Properties:

- Easy to implement
- Does not scale well
- Central server may fail

Slide 47

Properties:

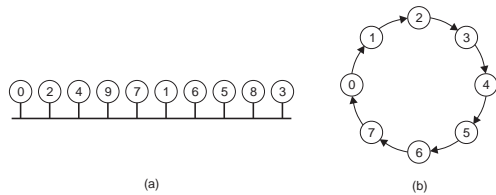
- Ring imposes an average delay of $N/2$ hops (limits scalability)
- Token messages consume bandwidth
- Failing nodes or channels can break the ring (token might be lost)

METHOD 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

Slide 46



METHOD 3: USING MULTICASTS AND LOGICAL CLOCKS

Algorithm by Ricart & Agrawala:

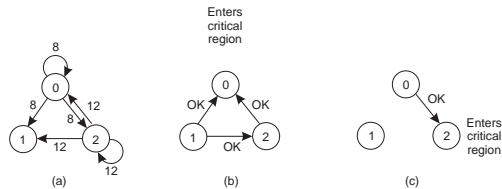
- Processes p_i maintain a Lamport clock and can communicate pairwise
- Processes are in one of three states:
 1. **Released**: Outside of critical section
 2. **Wanted**: Waiting to enter critical section
 3. **Held**: Inside critical section

Slide 48

Slide 49

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



Slide 50

Properties:

- Multicast leads to increasing overhead (more sophisticated algorithm using only subsets of peer processes exists)
- Susceptible to faults

EVALUATING DISTRIBUTED ALGORITHMS

General Properties:

- Performance
 - number of messages exchanged
 - response/wait time
 - delay
 - throughput: $1/(delay + executiontime)$
 - complexity: $O()$
- Efficiency
 - resource usage: memory, CPU, etc.
- Scalability
- Reliability
 - number of points of failure (low is good)

Slide 51

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: $2(n - 1)$

Slide 52

Reliability:

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