Synchronization

Discuss how processes can synchronize

For example, agree on the ordering of events, or avoid accessing a shared resource simultaneously

---

Topics to be covered

Clock Synchronization
Logical Clocks
Global State
Election Algorithms
Mutual Exclusion
Distributed Transactions

Model

Assume we have N processes p_i (i = 1, 2, …, N)

Each process

- executes on a single processor
- has a state that changes as it executes
- executes a series of actions (either a message send or receive operation or an internal operation of the process (e.g., update of one of its variables)

Event: the occurrence of a single action

Events within a single process p_i can be placed in a **single total order** $\rightarrow_i$

Each process is characterized by its history, a series of events that occur at each process.

$$h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

s_i^0: **initial state**

Clock Synchronization

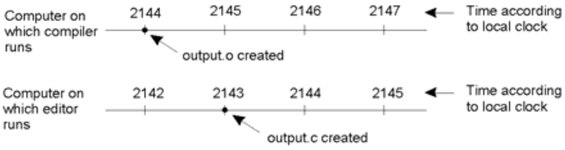
Physical Clocks
Cristian's Algorithm
The Berkeley Algorithm

Clock Synchronization

In a centralized system, time is unambiguous.

In a distributed system, achieving agreement on time is not trivial

Example (make)



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical Clocks

- Is it possible to synchronize all clocks in a distributed system?

Each computer has a circuit for keeping track of time

clock – timer a quartz crystal, when kept under tension, quartz crystals oscillate at a well-defined frequency

A counter & holding register: the counter is decremented by one at each crystal oscillation, when it gets to zero, an interrupt (clock tick) the counter is reloaded from the register

Can be programmed to give an interrupt say 60 times a sec

(software) clock: each interrupt adds 1 to the time stored in memory

With a single computer and a single clock, does not matter if the clock is off by a small amount – all processes use the same clock

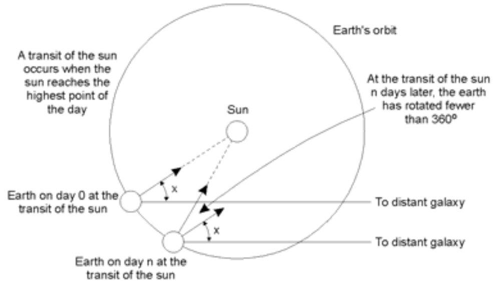
Clock skew: difference in time values between the software clocks

---

Physical Clocks

- How time is actually measured: Astronomically

Transit of the sun: sun reaching its highest apparent point in the sky
Solar day: interval between two consecutive sun transits. Solar second 1/864000 of a solar day
However, the period of the earth's rotation is not constant, mean solar second



Computation of the mean solar day.

Physical Clocks

- How time is actually measured: Atomic Time: Counting transitions of the cesium 133 atom

Universal Coordinated Time (UTC)

Based on the number of transitions per second of the cesium 133 atom (1 sec = time it takes to make 9,192,631,770 transitions

At present, the real time is taken as the average of some 50 cesium-clocks around the world

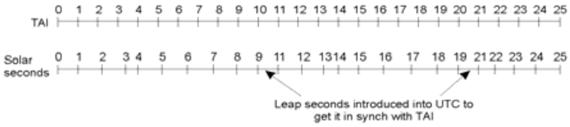
Introduces a leap second from time to time to compensate that days are getting longer

UTC is broadcasted through short wave radio (WWV receivers) and satellite. Satellites can give an accuracy of about ±0.5 ms

Does this solve all our problems?

Physical Clocks

UTC = TAI with leap seconds



TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

Clock Synchronization Algorithms

Each machine has a **timer** that causes an interrupt **H** times per second

A **(software) clock** keeps track of the number of ticks (interrupts) since some agreed-upon time in the past.

When the timer goes off, the interrupt handler adds 1 to the software clock

Let C be the value of the clock. Specifically, if UTC time is t, let the value of the clock on machine p be $C_p(t)$

Perfect world, $C_p(t) = t$ for all p and t, dC/dt = 1

Theoretically, a timer with H = 60, generate 216,000 (= 24*60*60) ticks per hour

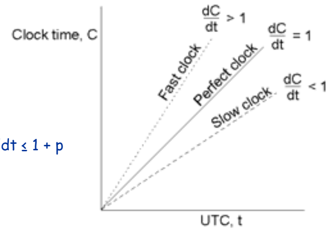
Real world, relative error 10^{5}, 215,998 to 216,002 ticks per hour

Maximum drift rate p:

$$1 - p \leq dC/dt \leq 1+p$$

Clock Synchronization Algorithms

The relation between clock time and UTC when clocks tick at different rates.



$1 - p \leq dC/dt \leq 1 + p$

- If two clocks, drift in the opposite direction, max 2p Δt apart
- No clocks differ more than δ: resynchronize (in software) at least every δ/2p

Physical Clocks

- How to synchronize clocks

Internal synchronization: Synchronize them with each other

For a synchronization bound $D > 0$, $|C_i(t) - C_j(t)| < D$

External Synchronization: synchronize them with real world clocks, say a source S of UTC time.

For a synchronization bound $D > 0$, $|S(t) - C_i(t)| < D$

- If a system is externally synchronized with bound D then it is internally synchronized with bound 2D

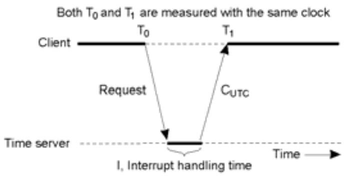
Clock Synchronization Algorithms

Cristian's Algorithm

There is a time server (WWV receiver)

Goal: have all other machines synchronized with it

1. Periodically with period $T < \delta/2p$, each machine asks the time server for the current time
2. The server responds asap with the current time, C_{UTC}
3. The client set its clock to C_{UTC}



Clock Synchronization Algorithms

Problems

1. Time must never run backwards, why? (Monotonicity condition)

Introduce changes gradually

2. It takes a nonzero amount of time for the time server's reply gets back to the sender

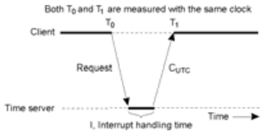
Measure it, best estimate $(T_1 - T_0)/2$

If the interrupt handling time, I, is known, $(T_1 - T_0 - I)/2$

Make a series of measurements

Any measurements in which $T_1 - T_0$ exceeds some threshold value are discarded

Average the estimations, or the faster messages are the most accurate



Clock Synchronization Algorithms

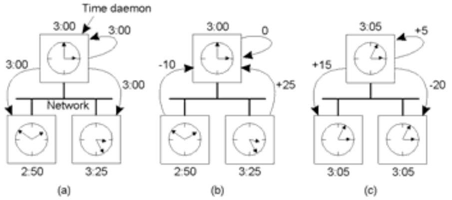
The Berkeley Algorithm

1. A time deamon periodically polls every machine to ask the time
2. Each machine replies
3. Based on the answers, computes an average. Informs every machine to advance or slow down its clock

The time daemon asks all the other machines for their clock values

The machines answer

The time daemon tells everyone how to adjust their clock



Clock Synchronization Algorithms

A Decentralized Algorithm

Divide time into fixed-length (R) resynchronization intervals

i-th interval: $[T_0 + iR, T_0 + (i + 1)R)$, T_0 some agreed-upon time instance in the past

Each machine:

1. At the beginning of each interval, broadcasts its current time (note, these broadcasts will not happen precisely simultaneously, why?)
2. Starts collecting all other broadcasts that arrive during an interval S
3. Runs an algorithm (e.g., average; discard m highest and m lower values and average the rest) to compute a new time from them

---

Use of Synchronized Clocks

New algorithms that utilize synchronized clocks

Example: Enforcing at-most-once message delivery, even in the face of crashes

Traditional approach: each message bears a unique message number (the server store all message number it has seen. Problem, if the server crashes and reboots, also how long to keep message numbers)

Modified approach: each message carries a connection identifier (chosen by the server) + a timestamp (its local time)

For each connection (i.e., sending process), the server records the most recent timestamp (that is, the largest timestamp) it has seen

Any incoming message for a connection with a timestamp that is lower than the stored timestamp is rejected as duplicate

To determine, when to renove a timestamp, each server maintains a variable G

G = CurrentTime – MaxLifeTime – MaxClockSkew

MaxLifeTime (how long after its transmission a message arrives)

MaxClockSkew (synchronization bound among clocks)

Wite G to disk every Δt

Logical Clocks

Lamport Timestamps
Vector Timestamps

Lamport Timestamps

It suffices that two processes agree on the order in which events occur (no need to synchronize their clocks)

The happens-before relation

a **happens-before** b, **a → b**: means that each process agrees that first event a occurs, then afterwards event b occurs

Two cases, where happens-before can be directly observed:

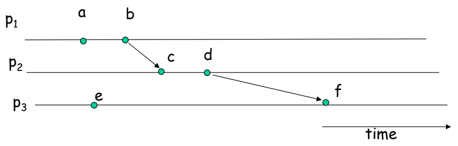
1. If ∃ process p_i: a $\rightarrow_i$ b, then a → b (that is if a and b are events in the same process, and a occurs before b then a → b is true)
2. If a is the event of a message being sent by one process and b is the event of the message being received by another process, then a → b is true. (For any message m, send(m) → receive(m))

Transitive relation, If a → b and b → c, then a → c.

If e and e' are events, and if e → e', then we can find a series of events e_1, e_2, …, e_n occurring at one or more processes such that e_1 = e and e' = e_n and for i = 1, 2, …, n, either case 1 or case 2 applies between e_i and e_{i+1} (that is, either they occur in succession in the same process, or there is a message m such that e_i = send(m) and e_{i+1} = receive(m))

- The sequence of events e_1, e_2, …, e_n may not be unique.

Example:



Case 1: a → b, c → d, e → f Case 2: b → c, d → f. What about a and e?

Two events, a and b, such that neither a → b nor b → a holds are said to be **concurrent** (happens-before is a partial order)

Goal: For every event a, assign a time value L (Lamport timestamp) such that all processes agree on it

Property of L: If a → b, then L(a) < L(b)

L must always go forward (increasing)

Algorithms for assigning timestamps to events

Each process p_i maintains its own logical clock L_i.

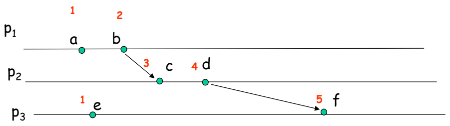
A Lamport logical clock is a monotonically increasing counter used to apply Lamport timestamps to events. (we denote them L_i(e) or L(e)).

1. L_i is incremented before each event is issued: $L_i = L_i + 1$
2.

(a) When a process sends a message m, it also sends a timestamp t = L_i

(b) When a message (m, t) arrives at the receiver process p_j, then p_j sets $L_j = \max(L_j, t)$ and before timestamping the event receive(m) applies rule 1

Example

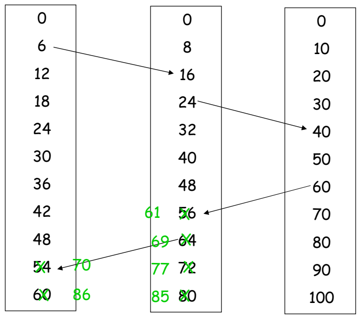


It can be shown that:

For any two events a and b, a → b ⇒ L(a) < L(b)

The converse in not true. For instance in the example, L(b) > L(e) but b and e are concurrent

---



No need to increment by 1, but any positive number

Lamport Timestamps

Totally ordered logical clocks

An additional requirement, no two events have numerically identical Lamport timestamps Attach the number (identifier) of the process in which the event occurs at the timestamp.

For instance, the low-order end of time separated by a decimal point e.g., 40.1 or 40.2

In general: $L_i(e).i$

Thus for all distinct events, a and b, $L(a) \neq (b)$

Totally-Ordered Multicast

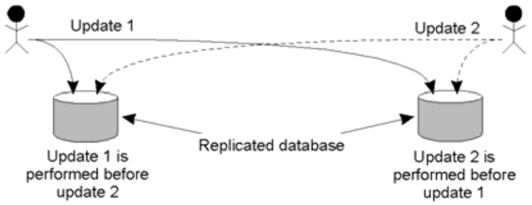
Example: a database replicated across several sites

Issue: update operations must be performed in the same order at each copy, so that all copies are exactly the same

Example:

Account = 1000, p_1 adds 100, p_2 increments by 1%

Replica1 1111 Replica2 1110



Totally-Ordered Multicast

Requirement of a totally-ordered multicast: a multicast operation by which all messages are delivered in the same order to each receiver

Assumption: reliable FIFO delivery of messages

Process p_i **sends** timestamped messages (with the current logical clock of the receiver), msg_i, to all others. Puts message in a local queue $queue_i$

Process p_j **receives** msg_i

Puts it onto a local queue $queue_j$ ordered according to its timestamp

Multicasts an acknowledgment (note, the timestamp of the received message is lower than the timestamp of the acknowledgement)

A process p_j can **deliver** a queued message msg_i to an application, only when:

(1) the message is at the head of the $queue_j$

(2) For each process p_k there is a message msg_k in $queue_j$ with a larger timestamp (i.e., the message has been acknowledged by each other process)

Vector Clocks

Vector Clocks

Goal: overcome the fact that we cannt conclude the order of events from the values of their timestamps, that is, from $L(a) < L(b)$, we cannot conclude that $a \rightarrow b$

A vector clock for a system of N processes is an array of N integers.

Each process keeps each own **vector clock** Vi which it uses to timestamp local events. Processes add vector timestamps on the messages they send.

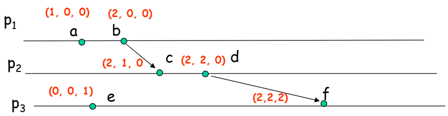
1. Initially, $V_i[j] = 0$, for i, j = 1, 2, …, n
2. Just before p_i timestamps an event, it sets $V_i[i] = V_i[i] + 1$
3. p_i includes the value $t = V_i$ in every message it sends (the whole vector)
4. When p_i receives a message with timestamp t, it sets $V_i[j] = \max(V_i[j], t[j])$ for j = 1, 2, n (that is, it takes, the component-wise maximum of two vector timestamps, known as a merge operation)

For a vector clock V_i, $V_i[i]$ is the number of events that p_i has timestamped and $V_i[j]$ for $i \neq j$ is the number of events that have occurred at p_j that p_i has potentially been affected by

---

Vector Clocks

Example



How to compare vector timestamps:

$V = V'$ iff $V[j] = V'[j]$ for j = 1, 2, …, n

$V \leq V'$ iff $V[j] \leq V'[j]$ for j = 1, 2, …, n

$V < V'$ iff $V[j] \leq V'[j]$ and $V \neq V'$

It can be shown that:

For any two events a and b, $a \rightarrow b \Rightarrow L(a) < L(b)$

The converse also holds, $L(a) < L(b) \Rightarrow a \rightarrow b$

For instance in the example, b and e are concurrent which can be also concluded by the fact that neither $V(e) \leq V(b)$ nor $V(b) \leq V(e)$

Global State

Global state = Local state of each process + messages currently in transit

How to ascertain a global state in the absence of global time?

If all processes had perfectly synchronized clocks, then agree on a time that each process would record each state, but …

Model

Assume we have N processes p_i (i = 1, 2, …, N)

Characterize each process by its **history**, a series of events that occur at each process.

$$h_i = <e_i^0, e_i^1, e_i^2, \ldots>$$

Finite prefix of the history

$$h_i^k = < e_i^0, e_i^1, \ldots, e_i^k>$$

event: an internal action of the process (e.g., update of one of its variables) or sending or receipt of a message

state of a process p_i, s_i^k, the state of process immediately after the kth event occurred

s_i^0: **initial state**

Global history

$$H = h_0 \cup h_1 \cup .. \cup h_{N\;1}$$

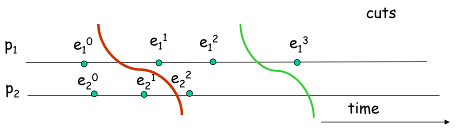
Global State (or **distributed snapshot**)

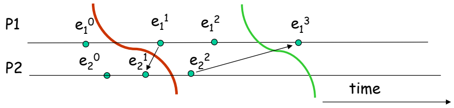
Which states are meaningful, which combination of process states could have occurred at the same time?

Corresponds to initial prefixes of the individual process histories

A **cut** of the system's execution is a subset of its global state that is a union of prefixes of process histories

$$C = h_0^{c1} \cup h_1^{c2} \cup .. \cup h_{N\;1}^{cn}$$





Are all cuts acceptable?

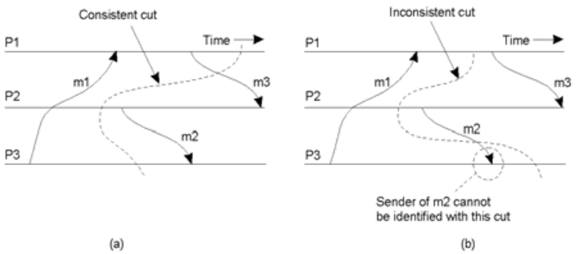
Say e_1^1 is the sending of a message and e_2^1 is the receipt

The actual execution never was in a global state corresponding to the process states at that frontier, examine the relation about events

A cut C is **consistent** if, for each event it contains, it also contains all the events that happened-before that event,

For all events $e \in C$, if $f \rightarrow e$, then $f \in C$

More examples



(a)

(b)

A **consistent global state** is one that corresponds to a consistent cut

The **execution** of a distributed system as a transition between global states of the system

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$$

In each transition, precisely one event occurs at some single process of the system

- A **run** is a **total** ordering of all events in a global history that is consistent with each local history's ordering
- A **consistent run** or **linearization** is an ordering of the events in a global history that is consistent with the happened-before relation on H

Not all runs pass through consistent global states, but all linearizations do

A state S' is **reachable** from state S if there is a linearization that passes through S and S'

Global State Predicates, stability, safety and liveness

Testing for properties amounts for evaluating a global state predicate

A global state predicate is a function that maps from the set of global states of processes in the system to {True, False}

Stable properties: once True at a state, remain True for all future states reachable from that state

Two interesting properties:

Suppose α is an undesirable property (e.g., deadlock)

Safety with respect to α is the assertion that α evaluates to False for all states S reachable from S_0.

Conversely, let β be a desirable property (e.g., reaching termination)

Liveness with respect to β is the property that, for any linearization L starting in state S_0, β evaluates to True for some state S_L reachable from S_0

The Chandy and Lamport Snapshot Algorithm

Goal: record a set of process and channel states

If a message has been sent by a process P but not received by a process Q, we consider it part of the channel between them

Assumptions:

- Neither channels nor processes fail
- Reliable communication, any message sent is received exactly once
- Unidirectional channels, FIFO-ordered message delivery
- There is a path between any two processes
- The processes may continue their execution and send and receive messages while the snapshot algorithm takes place

---

The Chandy and Lamport Snapshot Algorithm

Any process, say P, initiates the algorithm:

P records its own state

P sends a marker along each of its outgoing channels

Process Q:

When Q receives a marker through incoming channel C

If it has not saved its local state,

Records it, starts recording all incoming messages

Sends a marker along each of its outgoing channels

Else,

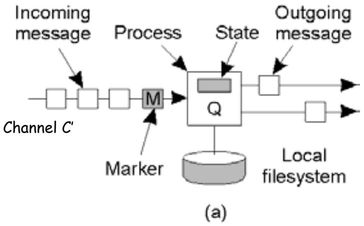
Stops recording the state of channel C (**state of C from R to Q**: Q records any message on C that arrived after Q recorded its state and before the sender (R) recorded its own state)

Finishes when it has received and processed a marker along each of its incoming channels

---

Example

Q receives marker for first time

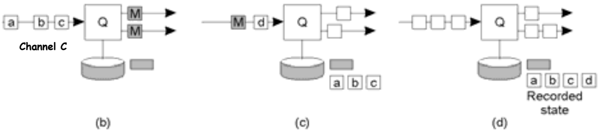


Example (continued)

Q records its local state and sends markers along each of its outgoing edges

Q records all incoming messages

Q finishes recording the state of incoming channel



Note

Records a consistent state but one that may never have occurred at the same time

Termination of the snapshot algorithm

Proof

We assume that a process that has received a marker records its state within a finite time and send markers over each outgoing channel within a finite time.

If there is a path of communication channels and processes from p_i to p_j, then p_j will record its state a finite time after p_i recorded its state

Since the graph is strongly connected, it follows that all processes will record their states and the states of their incoming channels a finite time after some process initially records its state

---

The algorithm selects a cut from the history of execution

We shall prove that this cut is consistent

Proof

Let e_i and e_j be events occurring at p_i and p_j respectively such that $e_i \rightarrow e_j$

We need to show that if e_j is in the cut then e_i is also in the cut

For the purposes of contradiction, assume that *e_i is not in the cut*, that is, p_i recorded its state before e_i occurred

Let m1, m2, …, mk the sequence of messages that lead to $e_i \rightarrow e_j$

By FIFO ordering, the marker from p_i would have reached p_j before these messages, thus p_j would have recorded its state before event e_j

This contradicts our assumption that p_j is in the cut.

We shall prove a reachability relation between the observed global state and the initial and final states when the algorithm runs

Let

S_{init}: the global state immediately before the first process recorded its state

S_{final}: the global state when the snapshot algorithm terminates (immediately after the last state recording action)

S_{snap} the recorded global state

Sys = $e_0, e_1, \ldots$ a linearization of the system as it executed (actual execution)

We shall show that there is a permutation of Sys, Sys' = $e'_0, e'_1, e'_2, \ldots$ such that all three states, S_{init}, S_{snap} and S_{final} occur in Sys'

Proof.

Categorize all events in Sys as pre-snap and post-snap events

A pre-snap event at process p_i is one that occurred at p_i before pi recorded its state. All other post-snap.

(Note a post-snap event may occur before a pre-snap event in Sys, if the two events belong to different processes)

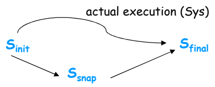
Suppose e_j is a post-snap event at one process and e_{j+1} is a pre-nap event at a different process:

It cannot be that $e_j \rightarrow e_{j+1}$ (why?)

Thus, we can swap the two events without violating the happened-before relation

We continue swapping until all pre-snap events $e'_0, e'_1, e'_2, \ldots. e'_{R-1}$ are ordered prior to all post-snap events $e'_R, e'_{R+1}, e'_{R+2}, \ldots$

$S_{snap} = e'_0, e'_1, e'_2, \ldots. e'_{R-1}$



Example:

Take a snapshot for detecting **termination of a computation**

How? Use the snapshot algorithm

When Q receives the marker for the first time, considers the process that sent that marker as its predecessor

When Q completes sends its predecessor a DONE message

When the initiator of the distributed snapshot receives a DONE from all its successors, the snapshot has been completely taken

Sends a DONE or a CONTINUE

When it sends a DONE?

- All of Q's successors have returned a DONE message
- Q has not received any message between the point it recorded its state, and the point it had received the marker along each of it incoming channels

Problem: incoming messages
We need a snapshot in which all channels are empty

Election Algorithms

The Bully Algorithm
A Ring Algorithm

Election Algorithms

Election algorithm: an algorithm for choosing a unique process to play a particular role, i.e., coordinator

All processes must agree on the choice

The Bully Election Algorithm

1. P sends an ELECTION message to all processes with higher numbers
2. If no one responds, P wins the election and becomes the coordinator
3. If one of the higher-ups answers, it takes over.

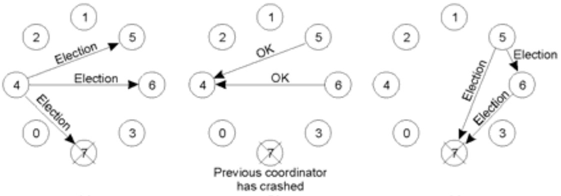
Assumes:

- Reliable message delivery, but processes may crash
- That the system is synchronous, assumes (timeouts to detect a process failure)
- Each process knows which processes have higher identifiers and can communicate with them

---

The Bully Election Algorithm

Example



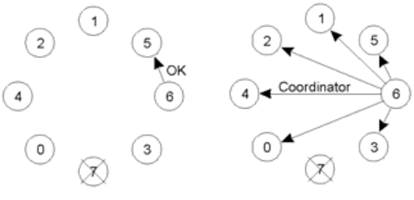
(a) (b) (c)

The bully election algorithm

- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

The Bully Election Algorithm

Example (continued)



(d) (e)

- Process 6 tells 5 to stop
- Process 6 wins and tells everyone

When 7 comes back, it holds an election

The Ring Election Algorithm

Assumption: each site knows its successor in the ring

1. Any site P may initiate the procedure.

2. Each site:

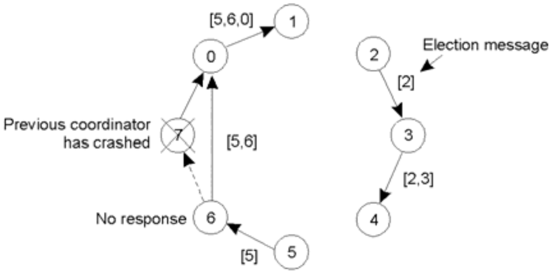
- Sends an ELECTION message to its successor, adds its number in the list
- If the successor is down, the sender skips over the successor and goes to the next member along the ring

3. When the message arrives at the initiating site P (how is this detected?) P circulates a COORDINATOR message with the higher number in the list as the coordinator

---

The Ring Election Algorithm

Example



Two simultaneous elections

Mutual Exclusion

A Centralized Algorithm
A Distributed Algorithm
A Token-Ring Algorithm

Mutual Exclusion

To read or update shared data structures, enter a critical region (CR) to achieve mutual exclusion

In centralized systems: semaphores, monitors, etc

Mutual Exclusion

Essential requirements for mutual exclusion:

Safety: At most one process may execute in the CR at a time

Liveness: Requests to enter and exit the CR eventually succeed

Liveness implies freedom of deadlocks and starvation (indefinite postponement of entry for a process that has requested it)

Absence of starvation is a fairness condition.

Another fairness conditions: order in which process enter the CR

The order that process enter the CR follows their requests to enter the CR:

If one request to enter the CR happened-before another, then entry to the CR is granted in that order

---

A Centralized Mutual Exclusion Algorithm

Select one process as the coordinator

- To enter a CR, sent a **<request>** message to the coordinator
- If no other process in the CR, the coordinator sends a **<grant>** message

Else, denies permission (e.g., does not reply and thus blocks the requesting process, or send a deny message)

- Upon exiting a CR, send a **<release>** message to the coordinator. The coordinator grants access to another process (e.g., takes the first item of the queue and sends a grant message)

Correct (safety): Guarantees mutual exclusion?

Fair: No starvation? Order?

Easy to implement

But: the coordinator is a single point of failure & a performance bottleneck/no way to distinguish a dead coordinator from "permission denied"

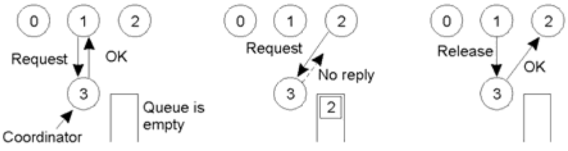
A Centralized Mutual Exclusion Algorithm

Example

Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

When process 1 exits the critical region, it tells the coordinator, when then replies to 2



(a) (b) (c)

A Decentralized Mutual Exclusion Algorithm

Ricart and Agrawala's algorithm

- Requires that there be a total order of all events in the system

(this can be achieved by using for example the Lamport's algorithm for providing timestamps)

- Assumes reliable sending of messages (i.e., every message is acknowledge)

A Decentralized Mutual Exclusion Algorithm

When a process wants to enter the CR,

- builds a **<request>** message M = (CR-id, process-number, timestamp)
- sends the message to all other processes (including itself)

Upon receipt of a <request> message M

i. If the receiver is not in the CR and does not want to enter the CR, replies **<OK>**

ii. If the receiver is in the CR, it does nor reply, queues M

iii. Else (the receiver is not in the CR, but wants to enter the CR),

Compares the timestamp with the timestamp of its own request,

if lower, replies <OK>, else does not reply, queues M

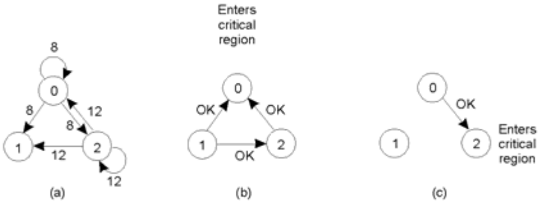
Waits till it receives OK from all processes

Upon exit from a CR,

- sends OK to all processes in its queue
- deletes them from the queue

A Distributed Mutual Exclusion Algorithm

Example



a) Two processes want to enter the same critical region at the same moment.
b) Process 0 has the lowest timestamp, so it wins.
c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

A Decentralized Mutual Exclusion Algorithm

Correct: guarantees mutual exclusion

No deadlock or starvation

However, worst than the centralized solution:

- Number of messages: 2(n-1)
- N points of failures! If a process fails, all others are blocked

Solution?

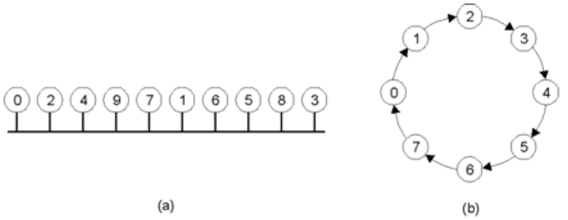
- Each process must maintain a list with all other processes
- Load balancing?

Slight improvement: Enter the CR, when granted permission from the majority (to work, a process after granting permission to a process, cannot grant permission to another one)

A Token-Ring Mutual Exclusion Algorithm

Construct a logical ring in which each process is assigned a position in the ring.

Each process knows who is next.



a) An unordered group of processes on a network.
b) A logical ring constructed in software.

A Token-Ring Mutual Exclusion Algorithm

When the ring is initialized, process 0 is given a token.

The token circulates the ring

When a process k acquires the token:

If it wants to enter the CR,

it enters the CR, does all the work, leaves the region,

passes the ring to k+1

Else,

it just passes the ring to k+1

A Token-Ring Mutual Exclusion Algorithm

Correctness (safety)?

Starvation?

Problems:

Lost token

Process crashes: require acknowledging the receipt of a token

Comparison

Algorithm	Messages per entry/exit	Client delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Messages per entry/exit determine the bandwidth consumed

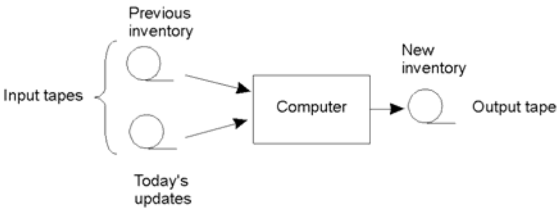
System throughput (the rate at which the collection of processes as a whole can access the critical region).

It is based on the synchronization delay between one process exiting the critical region and the next process entering it (not shown in the Table above)

Distributed Transactions

The Transaction Model
Classification of Transactions
Implementation
Concurrency Control

The Transaction Model



Updating a master tape is fault tolerant.

The Transaction Model

Examples of primitives for transactions.

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

The ACID properties

The Transaction Model

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
```

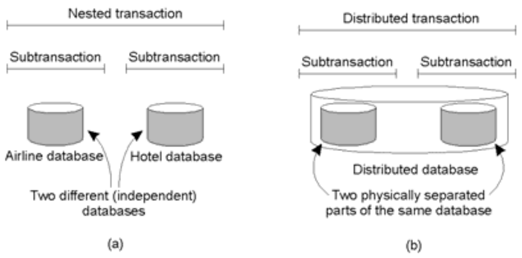
(a)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

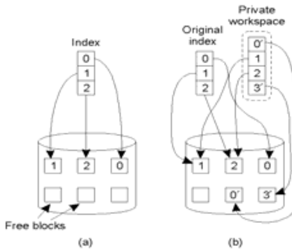
(b)

a) Transaction to reserve three flights commits
b) Transaction aborts when third flight is unavailable

Classification of Transactions



Implementation



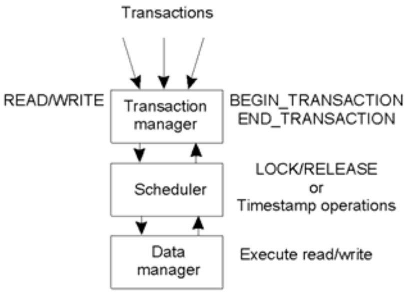
a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing

Implementation

	Log	Log	Log
x = 0;			
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

a) A transaction
b) – d) The log before each statement is executed

Concurrency Control



Transactions

READ/WRITE

Transaction manager

BEGIN_TRANSACTION
END_TRANSACTION

Scheduler

LOCK/RELEASE
or
Timestamp operations

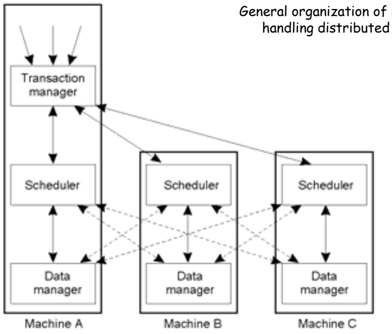
Data manager

Execute read/write

General organization of managers for handling transactions.

Concurrency Control

General organization of managers for handling distributed transactions.



Concurrency Control

BEGIN_TRANSACTION
x = 0;
x = x + 1;
END_TRANSACTION

(a)

BEGIN_TRANSACTION
x = 0;
x = x + 2;
END_TRANSACTION

(b)

BEGIN_TRANSACTION
x = 0;
x = x + 3;
END_TRANSACTION

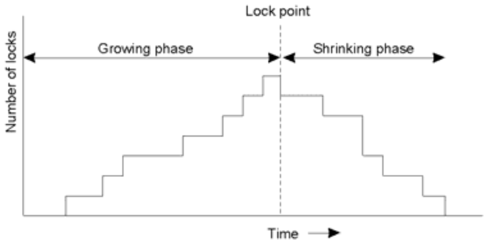
(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

(d)

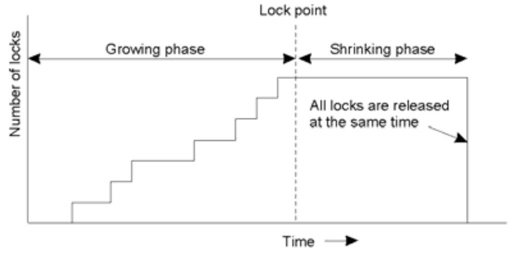
a) – c) Three transactions T_1, T_2, and T_3
d) Possible schedules

Concurrency Control



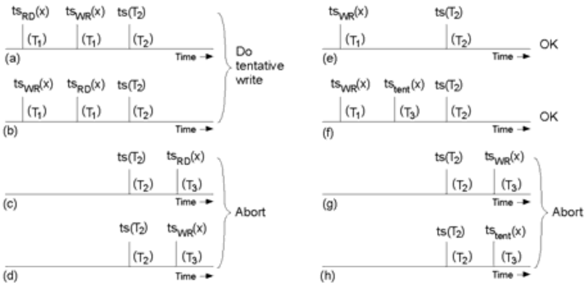
Two-phase locking.

Concurrency Control



Strict two-phase locking.

Concurrency Control



Concurrency control using timestamps.