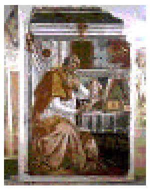
Chapter 6: Distributed Synchronization and Mutual Exclusion

- What is time? Do we have a global time in a distributed system?
- Synchronization with respect to physical time.
- Synchronization with respect to logical time.
- Distributed coordinator.
- Distributed mutual exclusion.

What is time?

- **an instance or single occasion for some event; "this time he succeeded"; "he called four times"; "he could do ten at a clip"**
- **an indefinite period (usually marked by specific attributes or a ctivities); "he waited a long time"; "the time of year for planting"; "he was a great actor is his time"**
- **a period of time considered as a resource under your control and sufficient to accomplish something; "take time to smell the roses"; "I didn't have time to finish"; "it took more than half my time"**
- **a suitable moment; "it is time to go"**
- **the continuum of experience in which events pass from the future through the present to the past**
- **clock time: the time as given by a clock; "do you know what time it is?"; "the time is 10 o'clock"**
- **clock: measure the time or duration of an event or action or the person who performs an action in a certain period of time; "he clocked the runners"**
- **fourth dimension: the fourth coordinate that is required (alongwith three spatial dimensions) to specify a physical event**
- **assign a time for an activity or event; "The candidate carefully timed his appearance at the disaster scene"**
- **a person's experience on a particular occasion; "he had a time holding back the tears"; "they had a good time together"**
- **set the speed, duration, or execution of; "we time the process to manufacture our cars very precisely"**
- **regulate or set the time of; "time the clock"**
- **meter: rhythm as given by division into parts of equal time**
- **prison term: the period of time a prisoner is imprisoned; "he served a prison term of 15 months"; "his sentence was 5 to 10 years"; "he is doing time in the county jail"**
- **adjust so that a force is applied an an action occurs at the desired time; "The good player times his swing so as to hit the ball squarely"**

St. Augustine's Dilemma:



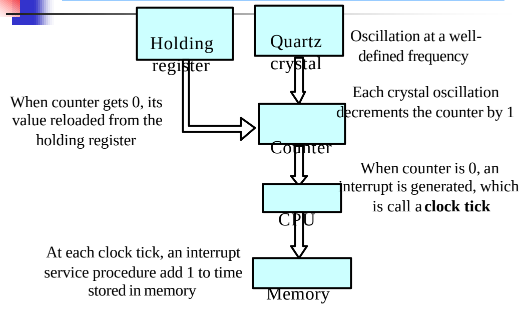
St. Augustine
354-430

"What then, is time?
If no one asks me,
I know.
If I wish to explain it to
someone who asks,
I know it not."

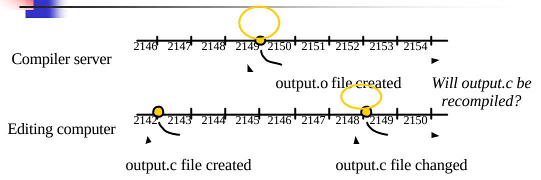
Physical Clock

- Also called Timer, usually a quartz crystal, oscillating at a well-defined frequency. A timer is associated with two registers: a counter and a holding register, and counter decreasing one at each oscillation.
- When the counter gets to zero, an interruption is generated and is called one clock tick.
- Crystals run at slightly different rates, the difference in time value is called a clock skew.
- Clock skew causes time-related failures.

How Clocks Work in Computer


Holding register
Quartz crystal
Oscillation at a well-defined frequency
Each crystal oscillation decrements the counter by 1
When counter gets 0, its value reloaded from the holding register
Counter
When counter is 0, an interrupt is generated, which is call a **clock tick**
CPU
At each clock tick, an interrupt service procedure add 1 to time stored in memory
Memory

Clock Skew problem



Why clocks need to be synchronised

- Many applications rely on correct and accurate timing
  - Real time applications, e.g. calculation of interests,
  - Version control and configuration management, e.g. the *make* command in Unix
- Correct and accurate clocks can simplify communication and concurrent control

What’s Different in Distributed Systems

- In centralized systems, where processes can share a clock and memory, implementation of synchronization primitives relies on shared memory and the times that events happened.
- In distributed system, processes can run on different machines.
  - No shared memory physically exists in a multi-computer system
  - No global clock to judge which event happens first
  - Logical simulation of shared memory is not a good solution

Synchronization With Physical Clocks

- How do we synchronize physical clocks with real-word clock?
  - TAI (International Atomic Time): Cs133 atomic clock
  - UTC (Universal Coordinated Time): modern civil time, can be received from WWV (shortwave radio station), satellite, or network time server.
  - ITS (Internet Time Service) NTS (Network Time Protocol)
- How do we synchronize clocks with each other?
  - Centralized Algorithm: Cristian’s Algorithm
  - Distributed algorithm: Berkeley Algorithm

How to synchronize local clock (1)

- If a local clock is slower, we can adjust it advancing forward (lost a few clock ticks), but how about it was faster than UTC?
  - set clock backward might cause time-related failures
- Use a soft clock to provide continuous time :
  - Let S be a soft clock, H the local physical clock
  - $S(t) = H(t) + \delta(t)$ (1)
  - The simplest compensating factor δ is a linear function of the physical clock: $\delta(t) = aH(t) + b$ (2)
  - Now, our problem is how to find constant a and b

How to synchronize local clock (2)

- Replace formula (1), we have
  - $S(t) = (1 + a)H(t) + b$ (3)
  - Let the value of S be T_{skew}, and the UTC at h be T_{real}, we may have that $T_{skew} > T_{real}$ or $T_{skew} < T_{real}$.
  - So S is to give the actual time after N further ticks, we must have:

    $T_{skew} = (1 + a)h + b$ (4)

    $T_{real} + N = (1 + a)(h + N) + b$ (5)
- Solve (4) and (5), we have:

  $a = (T_{real} - T_{skew})/N$

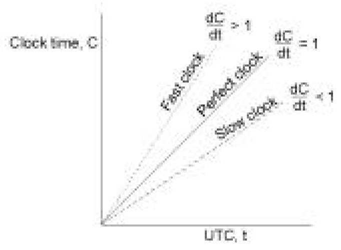
  $b = T_{skew} - (1 + a)h$

How to synchronize distributed clocks

- Assume at UTC time t, a physical clock time is H(t):
  - If they agree, then $dH/dt = 1$
  - But it is virtually impossible, for each physical clock, there is a constant ρ (given by manufacturers, called maximum drift rate), such that

    $1 - \rho \leq dH/dt \leq 1 + \rho$
  - If two clocks drift away from UCT in the opposite direction, then after Δt, they are $2\rho\Delta t$ apart.
  - Thus, if we want to guarantee that no two clocks ever differ by more than δ, clocks must be re-synchronized at least every $\delta/2\rho$ seconds.

Clocks Drifting



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

Cristian's algorithm (1)

Assumptions: There is a machine with WWV receiver, which receives precise UTC (Universal Coordinated Time). It is called the **time server.**

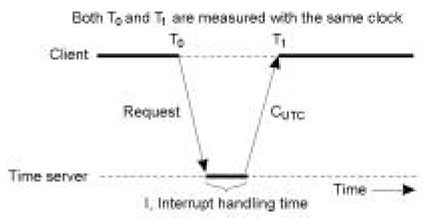
Algorithm:

1. A machine sends a request to the time server at least every $\delta/2\rho$ seconds, where δ is the maximum difference allowed between a clock and the UTC;

2. The time server sends a reply message with the current UTC when receives the request;

3. The machine measures the time delay between time serve's sending the message and the machine's receiving the message. Then, it uses the measure to adjust the clock.

Cristian's algorithm (2)


Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

Getting the current time from a time server

Cristian's algorithm (3)

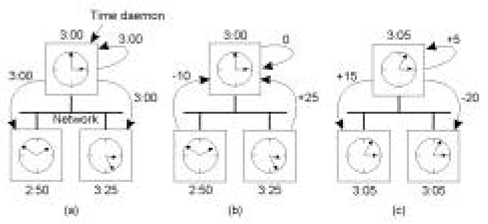
Measure the message propagation time

- (T1–T0)/2
- (T1–T0–I)/2
- Take a series of measures, and calculate the average;
- Take a series of measures, and use the fastest one.

Adjust the clock:

- If the local clock is faster than the UTC, add less to the time memory for each clock tick;
- If the local clock is slower than the UTC, add more to the time memory for each clock tick.

The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clocks

Logical Clock

A person with one watch knows what time it is.
A person with two or more watches is never sure.

- Lamport defined a relation called **happens before**, represented by →.
- The relation → on a set of events of a system is the relation satisfying three conditions:

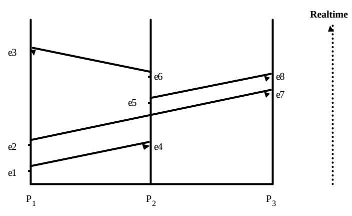
Conditions of Happens Before

- If **a** and **b** are events in the same process, and **a** comes before **b**, then **a** ® **b**.
- If **a** is the sending event of a message **msg** by one process, and **b** is the receipt event of **msg**, then **a** ® **b**.
- If **a** ® **b**, **b** → **c**, then **a** → **c** .

- Two distinct events **a** and **b** are concurrent if

a ? b and b ? a

Example: Event ordering



e1 ? e2 ? e3

e1 ? e4 ? e5 ? e6 ? e3

e2 ? e7 ? e8

e2 ? e4 and e5 ? e7

Logical Clock Condition

- For any events **a** and **b**, if **a** ® **b** then

  C(a) < C(b)
- From the definition of ®, the Clock Condition is satisfied if the following two conditions hold:
- Condition 1: if **a** and **b** are events in P_i, and **a** comes before **b**, then

  C_i(a) < C_i(b).
- Condition 2: if **a** is the sending of a **msg** by P_i and **b** is the receipt of the **msg** by P_j, then

  C_i(a) < C_j(b).

Implementation Rules

- IR1: Each process **P_i** increments **C_i** between any two successive events (for Condition 1).
- IR2: If event **a** is the sending of msg **m** by **P_i**, then **m** contains a timestamp **T_m = C_i(a).** Upon receiving a msg **m**, **P_j** sets **C_j** greater than or equal to **C_j** 's present value and greater than **T_m** (for Condition 2).

Total Ordering Relation

- If **a** is an event in **P_i** and **b** is an event in **P_j**, then

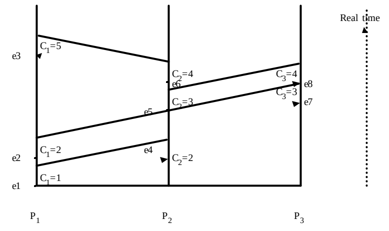
  a Þ **b** if and only if either:

  (1) **C_i(a) < C_j(b)** , or

  (2) **C_i(a) == C_j(b)** and **P_i < P_j** .

By Þ relation, we can totally order all events happened in a distributed system

Example: Event ordering *wrt* logical clocks



e1 Þ e2 Þ e4 Þ e5 Þ e7 Þ e6 Þ e8 Þ e3

Distributed Mutual Exclusion

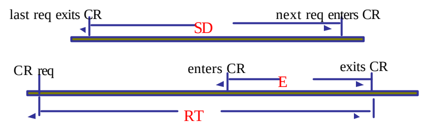
- Concurrent access to a shared resource (critical region) by several uncoordinated processes located on several sites is serialized to secure the integrity of the shared resource.
- The major differences comparing with the single processor ME problem are: (1) there is no shared memory and (2) there is no common physical clock.
- Two kinds of algorithms: logical clock based and token based.

Requirements of Distributed ME

- Mutual Exclusion: guarantee that only one request access the CR at a time.
- Freedom from deadlock: two or more sites(hosts) should not endlessly wait for msg's that will never arrive.
- Freedom from starvation: a site should not be forced to wait indefinitely to access CR.
- Fairness: requests must be executed in the order they are made. (fairness $\rightarrow$ freedom of starvation, but not reverse)
- Fault tolerance: in the case of failure, the algorithm can reorganize itself so that it continues to function without any disruption.

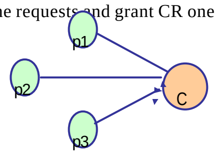
How to Measure Performance

- Number of msg's per CR invocation.
- Synchronization Delay (SD).
- Response Time (RT).
- System Throughput (ST):
- ST = 1/(SD + E)
- where E is the average CR execution time.



Centralized Solution

Queue up the requests and grant CR one by one.



- 3 msg's per CR invocation: REQ, ACK, REL
- Single point of failure
- Control site is a bottleneck
- SD = 2T where T is the communication delay
- ST = 1/(2T + E)

Distributed ME Algorithms

- S_i: a site (or a process)
- R_i: a Request Set, contains id's of all those sites from which S_i must acquire permission before entering CR.
- For example, the Centralized Solution:

  $R_i = \{S_C\}$

Lamport Algorithm

- $\forall i : 1 \le i \le n \quad R_i = \{S_1, S_2, \ldots S_n\}$
- Assumption: msg's to be delivered in the FIFO order between every pair of sites.
- Data structure: each site S_i maintains a Request queue, rq_i, which contains requests ordered by their timestamp.

Lamport Algorithm (A) Request:

- When S_i wants to enter the CR, it sends a

  REQ(t_{si}, i)

  to all sites in R_i and places the request on rq_i, where (t_{si}, i) is the timestamp of the request.
- When S_j receives the REQ(t_{si}, i) from S_i, it returns

  REPLY (t_{sj}, j)

  to S_i, and places S_i's REQ on rq_j

Lamport Algorithm (B) Enter CR:

- S_i enters the CR when the following two conditions hold:

  L1: S_i has receives a msg with timestamp later than (t_{si}, i) from all other sites.

  L2: S_i's request is at the top of rq_i.

Lamport Algorithm (C) Release:

- S_i, upon exiting the CR, removes its request from the top of rq_j , and sends a timestamped REL msg to all sites in R_i .
- When a site S_j receives a REL msg from S_i, it removes S_j's REQ from rq_j.

Correctness and Performance:

- Correctness: from the total ordering of timestamps, it is easy to prove that no two sites satisfy both L1 and L2 simultaneously.
- Performance:

  Number of msg's: $3(N - 1)$

  SD: T

Ricart-Agrawala Algorithm

- An optimization of Lamport's Algorithm.
- $\forall i : 1 \le i \le n \quad R_i = \{S_1, S_2, \ldots S_n\}$

Ricart-Agrawala Algorithm (A) Request:

- When S_i wants to enter the CR, it sends a

  $REQ(t_{si}, i)$

  to all sites in R_i .
- When S_j receives the $REQ(t_{si}, i)$ from S_i, it returns a

  $REPLY\ (t_{sj}, j)$

  to S_i if S_j is neither requesting nor executing the CR, or if S_j is requesting and S_i's REQ Þ S_j's REQ, otherwise places S_i's REQ on rq_j and the reply is deferred.

Ricart-Agrawala Algorithm (B) Enter CR:

- S_i enters the CR when the following condition holds:

  L: S_i has receives a REPLY from all other sites.

Ricart-Agrawala Algorithm (C) Release:

- S_i, upon exiting the CR, sends a timestamped REL msg to all sites with a deferred REQ .

Performance:

Number of msg's : 2(N – 1)
SD: T

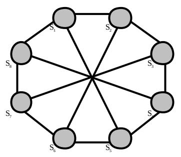
Maekawa Algorithm

- A site does not request permission from every other site, but only from a subset of the sites.
- A site locks all the sites in R_i in exclusive mode.
- Request set R_i is constructed to satisfy following conditions:
  M1: $\forall i\ \forall j\ \ i \neq j,\ 1 \leq i, j \leq n:\ R_i \cap R_j \neq \phi$
  M2: $\forall i\ \ 1 \leq i \leq n:\ S_i \in R_i$
  M3: $\forall i\ \ 1 \leq i \leq n:\ |R_i| = k$
  M4: $\forall i\ \forall j\ \ 1 \leq i, j \leq n$: any S_i is contained in k number of R_j's.
- Relationship between k and n: k = sqrt(n) + 1

Maekawa Algorithm: Comments

- At least one common site between R_i and R_j , from M1 and M2.
- All sites have to do an equal amount of work to invoke mutual exclusion, from M3.
- All sites have equal responsibility in granting permission to other sites, from M4.

Maekawa Algorithm: An example



$R_1 = \{ S_1, S_2, S_5, S_8 \}$
$R_2 = \{ S_1, S_2, S_3, S_6 \}$
$R_3 = \{ S_2, S_3, S_4, S_7 \}$
?

- take $R_5 = \{ S_4, S_5, S_6, S_1 \}$ and
  $R_8 = \{ S_7, S_8, S_1, S_4 \}$,
  we have $R_5 \cap R_8 = \{ S_4, S_1 \}$

Maekawa Algorithm (A) Request

- When S_i wants to enter the CR, it sends a
  REQ(t_{si}, i)
  to all sites in R_i .
- When S_j receives the REQ(t_{si}, i) from S_i, it returns a
  REPLY (t_{sj}, j)
  to S_i if S_j has not send a REPLY to any site from the time it received the last REL msg. Otherwise, it defers the REQ.

Maekawa Algorithm (B) Enter CR

- S_i enters the CR when the following condition holds:

  L: S_i has receives a REPLY from all sites in R_i.

Maekawa Algorithm (C) Release

- S_i sends a REL(i) to all sites in R_i.
- S_j, upon receiving a REL(i) , sends a REPLY to the next waiting REQ and delete that entry. If the waiting queue is empty, set NO_REPLY_SINCE_LAST_REL.

Correctness and Performance

- Correctness:

  Suppose that two sites S_i and S_j are concurrently in the CR. If $R_i \cap R_j = \{S_k\}$ then S_k must have send REPLY to both S_i and S_j concurrently, which is a contradiction to the role of

  NO_REPLY_SINCE_LAST_REL
- Performance:

  number of msg's: 3(sqrt (n) + 1)

  SD: 2T

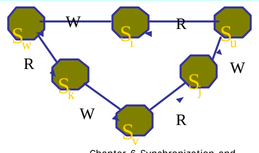
Problem: Potential of Deadlock

- Without the loss of generality, assume that three sites S_i, S_j and S_k simultaneously invoke Mutual Exclusion, and suppose:

  $R_i \cap R_j = \{S_u\}$

  $R_j \cap R_k = \{S_v\}$

  $R_k \cap R_i = \{S_w\}$
- Solution: extra msg's to detect deadlock, maximum number of msg's = 5(sqtr(n) + 1).



Token-based ME Algorithms

- A unique Token is shared among all sites.
- A site is allowed to enter the CR if it holds the Token.
- Token-based algorithms use a sequence number instead of timestamps.
- Correctness proof is trivial.
- Rather, the issues of freedom from starvation and freedom from deadlock are more important.

Suzuki-broadcasting Algorithm

- Distinguishing outdated REQ's from the current REQ.
- Determining which site has an outstanding REQ for CR.
- Data structure:

  REQ(j, n) : a request from S_j with sequencing number n.

  RN_j[1..n] : an array at S_j where RN_j[i] is the largest sequencing number received so far from S_i.

  Token { Q: REQ queue;

  LN[1..n] : where LN[j] is the most recent sequencing number of S_j .

  }

Suzuki Algorithm (A) Request

- If S_i does not hold the token, it increments its sequencing number, RN_i[i], and sends a

  REQ(i, RN_i[i])

  to all sites (broadcasting).
- When S_j receives the REQ(i, n) from S_i, it sets RN_j[i] to max(RN_j[i], n) . If S_j has the idle token, then it sends the token to S_i if RN_j[i] = LN[i] + 1 .

Suzuki Algorithm (B) and (C)

- (B) Enter CR:
- S_i enters the CR when it has the token

- (C) Release:
- S_i sets a LN[i] to $RN_i[i]$,
- For every S_j whose identifier is not in the token's Q, it appends S_j into in the token's Q if
  $RN_i[j] = LN[j] + 1$
- If the token's Q is non-empty after the above update, the S_i deletes the top identifier from the token's Q and sends the token to that identified site.

Correctness and Performance

- A requesting site enters the CR in finite time. Since one of the sites will release the token in finite time, site S_i's request will be placed in the token's Q in finite time. Since there can be at most n-1 requests in front of S_i, S_i will execute the CR in finite time.
- Performance:

number of msg's:	0 or n
SD :	0 or T

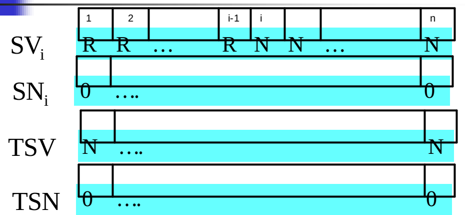
Singhal's Heuristic Algorithm

- Each site maintains information about the state of other sites and uses it to select a set of sites that are likely to have the token.
- A site must select a subset of sites such that at least one of those sites is guaranteed to get the token in the near future, otherwise, there is a potential deadlock or starvation.

Singhal Algorithm: Data Structure

- $\forall i : 1 \le i \le n$
  $SV_i[1..n]$: S_j's state array {R, E, H, N}.
  $SN_j[1..n]$: S_j's highest sequencing number array
- Token
  TSV[1..n] : Token's state array {R, N}.
  TSN[1..n] : highest sequencing number array
- States: a site can be in one of the following states:
  R: requesting the CR
  E: entering the CR
  H: holding the idle token
  N: none of the above

Singhal Algorithm: Initialization



- Property: for any S_i and S_j, either
  $SV_i[j] = R$ or $SV_j[i] = R$

Singhal Algorithm (A) Request

- If S_i does not hold the token:
  (1) $SV_i[i] \Leftarrow R$; (2) $SN_i[i] \Leftarrow SN_i[i] + 1$;
  (3) sends REQ(i, $SN_i[i]$) to all sites for which $SV_i[i] == R$.
- When S_j receives the REQ(i, m) from S_i, if $m \le SN_j[i]$ do nothing; otherwise $SN_j[i] \Leftarrow m$,
  cases:
  (1) If $SV_j[j] == N$, then $SV_j[i] \Leftarrow R$
  (2) If $SV_j[j] == R$ && $SV_j[i] != R$, then
    $SV_j[i] \Leftarrow R$, and sends REQ(j, $SN_j[j]$) to S_i
  (3) If $SV_j[j] == E$, then $SV_j[i] \Leftarrow R$
  (4) If $SV_j[j] == H$, then $SV_j[i] \Leftarrow R$,
    $TSV[i] \Leftarrow R$, $TSN[i] \Leftarrow m$, $SV_j[j] \Leftarrow N$
  and sends Token to S_i

Singhal Algorithm (B) and (C)

- (B) Enter CR: when S_i has the token, it sets $SV_i[i] == E$ and then enters the CR.

- (C) Release:
- If S_i finishes CR, it sets $SV_i[i] \Leftarrow N$, $TSV[i] \Leftarrow N$,
- For every S_j (j: 1..n) , if $SN_j[j] > TSN[j]$, then update token:

  $TSV[j] \Leftarrow SV_i[j]$, $TSN[j] \Leftarrow SN_j[j]$

  else update local:

  $SV_j[j] \Leftarrow TSV[j]$, $SN_j[j] \Leftarrow TSN[j]$
- If $(\forall j : SV_j[j] == N)$ then $SV_j[i] \Leftarrow H$, else selects a S_j such that $SV_j[j] == R$, and sends the token to that identified site.

Correctness and Performance

- See Simghal, M. “A Heuristically-aided Algorithm for Mutual Exclusion in Distributed Systems”, IEEE Trans on Computer, Vol. 38, No. 5, 1989
- Performance:

  number of msg’s: avarage n/2

  SD : T

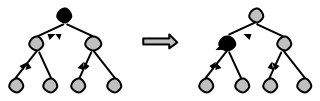
Raymond Tree-based Algorithm

- Sites are logically arranged as a directed tree such that the edges of the tree are assigned directions towards the root site that has the token.
- Data structure: for each S_i :

  holder: points to an immediate neighbour node on directed path to the root (which is self-pointed)

  RQ: stores requests received by S_i, but have not yet been sent the token. (An FIFO queue)

Raymond Algorithm: An Example



- Root transition when Token has been passed to another node in the tree.

Raymond Algorithm (A) Request

- If S_i does not hold the token and its RQ_i is empty, it sends a REQ(i) to holder, and appends the request to RQ_i .
- when S_j receives the REQ(i) , it places the REQ(i) in its RQ_j and sends a REQ(j) to holder provided it is not the root and its RQ_j has a single entry.
- when the root receives a REQ(k) , it sends the token to the sender S_k and redirect holder to the sender.
- when S_j receives the token, if the top entry in RQ_j is not its own request, it deletes the top entry, sends the token to the top entry site, and redirect holder to that site. If RQ_j is not empty at this point, then sends a REQ(j) to the new holder.

Raymond Algorithm (B) and (C)

- (B) Enter CR:
- when S_i has the token and its own request is on the top of RQ_i, then deletes the top entry and enters the CR.

- (C) Release:
- If S_i finishes CR and its RQ_j is not empty, it deletes the top entry, sends the token to the top entry site, and redirect holder to that site.
- If S_i’s RQ_j is not empty at this point, it sends a REQ(i) to holder .

Correctness and Performance

- Deadlock free: the acyclic nature of tree eliminates the possibility of circular wait among requesting sites.
- Starvation free: FIFO nature of request queue.
- Performance:

number of msg's:	O(log n)
SD :	T*log n/2

Comparison of Distributed ME Algorithms

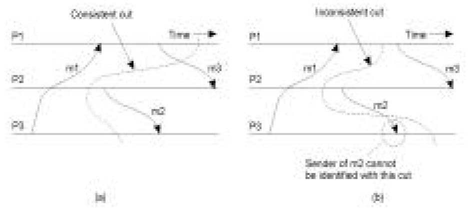
Algorithm	Response time	SD	# of messages (LL)	# of messages (HL)
Lamport	2T + E	T	3(n – 1)	3(n –1)
Ricart-Agrawala	2T + E	T	2(n – 1)	2(n – 1)
Maekawa	2T + E	2T	3(sqrt(n) + 1)	5(sqrt(n) + 1)
Suzuki - Kasami	2T + E	T	n	n
Sinhal	2T + E	T	n/2	n
Raymond	T(log n) + E	Tlog n/2	log n	4

LL: Light Load, HL: Heavy Load

Global State and Distributed Coordinator

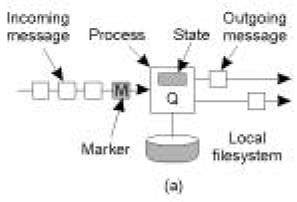
- The global state of a distributed system consists of the local state of each process, together with the messages that are concurrently in transit.
- The coordinator of a distributed system is a process (assigned or elected) which takes special responsibility and performs some special role.

Global State



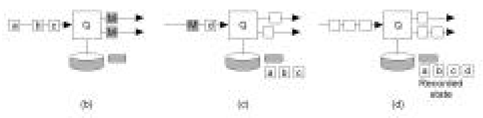
a) A consistent cut
b) An inconsistent cut

Global State: Distributed Snapshot(1)



a) Organization of a process and channels for a distributed snapshot

Global State: Distributed Snapshot(2)



b) Process Q receives a marker for the first time and records its local state
c) Q records all incoming message
d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel

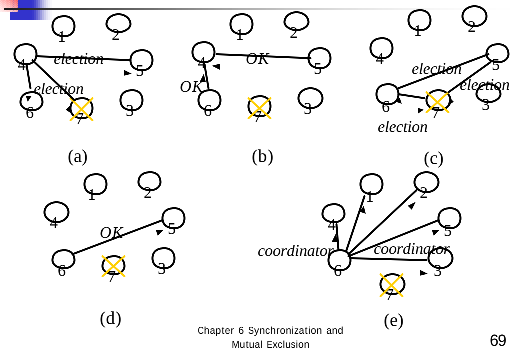
Election Algorithms

- Where a distributed algorithm requires a process to act as coordinator, an election algorithm can be invoked.
- The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.
- Assumptions:
  - Each process has a unique number, for example, its network address.
  - Every process knows the process number of every other process. What is unknown is which ones are currently up and which ones are currently down.
  - The election algorithm attempts to locate the process with the highest number and designates it as coordinator.

Election: The Bully Algorithm

- When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process *P* holds an election as follows:
  - *P* sends an *ELECTION* message to all processes with higher number;
  - If no one responds, *P* wins the election and announces that it is the new coordinator;
  - If one of the higher-ups answers, it takes over. *P*'s job is done.
- When a process gets an *ELECTION* message from one of its lower-numbered colleagues,
  - the receiver sends an *OK* message back to the sender,
  - it takes over the election, unless it is already the coordinator.
- If a process that was previously down comes back, it holds an election.

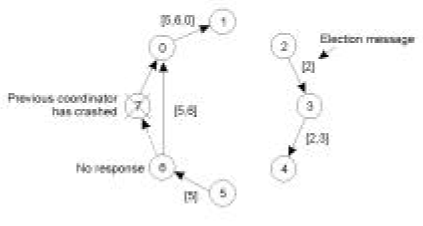
The Bully Algorithm: An Example



Election: A Ring Algorithm

1. The processes are logically organised as a ring
2. When a process notices that the coordinator is not functioning, it initiates election by send an ELECTION message to its successor.
   - The *ELECTION* message contains its number;
   - If the successor is down, the sender skips over the the successor and goes to the next member alone the ring until a running process is located.
3. When a process receives an *ELECTION* message, it checks if its own number is in the list of processes contained in the message,
   - If not, it inserts its number into the message and pass the message alone the ring.
   - if yes, the highest number in the list is elected as the coordinator, a *COORDINATOR* message is circulated, which contains who is the coordinator and who are the members of the ring.

A Ring Algorithm: An Example



Performance Analysis

- Number of messages:
  - Bully algorithm:
    - $(N^2 - 1)$
  - Ring algorithm:
    - $2N$, where N is the number of processes.
- Time delay:
  - Bully algorithm:
    - If broadcasting messages: 3T.
    - If no broadcasting messages: $(N+1)$T.
  - Ring algorithm:
    - $(N-1)$T