CS2056 DISTRIBUTED SYSTEMS

UNIT IV

Time and Global States-Introduction-Clocks, events and process states-Synchronizing physical clocks-Logical time and logical clocks-Global states-Distributed debugging.
Coordination and Agreement-Introduction-Distributed mutual exclusion-Elections-Multicast communication-Consensus and related problems.

• Importance of time in distributed systems
– A quantity to timestamp events accurately
    • To know what time a particular event occurs
    • i.e. Recording when an e-commerce transaction occurs
– A synchronization source for several distributed algorithms
    • To maintain consistency of distributed data
    • i.e. Eliminating duplicate updates
– A timing source for multiple events
    • To provide relative order of two events
    • i.e. Ensuring the order of cause and effect
    • Clocks in computers to establish
– *Time* at which an event occurred
– *Duration* of an event or interval between two events
– *Sequence* of a series of events or the order in which events occurred

11.2 Clocks, Events and Process States

• A distributed system consists of a collection *P* of ***N*** processes ***pi***, *i = 1,2,… N*
– Each process *pi* has a state ***si*** consisting of its variables (which it transforms as it executes)
– Processes communicate only by messages (via a network)
• **Actions** of processes*: Send, Receive,* change own state
• ***Event****:* the occurrence of a single action that a process carries out as it executes
• Events at a single process *pi*, can be placed in a total **ordering** denoted by the relation →*i* between the events. i.e.
– ***e*** →***i*** **e'** if and only if evente occurs before event *e'* at process *pi*
• A history of process pi: is a series of events ordered by →*i*
– ***history*****(pi)** = hi =<e*i*0, *ei*1, *ei*2, …>

Clocks

To timestamp events, use the computer's clock • At **real time, *t***, the OS reads the time on the computer's **hardware clock *Hi*(*t*)**
• It calculates the time on its **software clock *Ci*(*t*)=α*Hi*(*t*) + β**

– e.g. a 64 bit value giving nanoseconds since some base time
– **Clock resolution**: period between updates of the clock value

• In general, the clock is not completely accurate – but if *Ci* behaves well enough, it can be used to timestamp events at *pi*

Skew between computer clocks in a distributed system

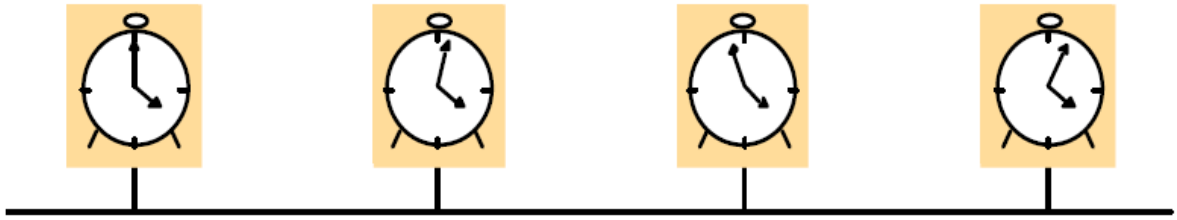


Figure 11.1

Computer clocks are not generally in perfect agreement
• ***Clock skew***: the difference between the times on two clocks (at any instant)
• Computer clocks use crystal-based clocks that are subject to physical variations
– ***Clock drift***: they count time at different rates and so diverge (frequencies of oscillation differ)
– ***Clock drift rate***: the difference per unit of time from some ideal reference clock
– Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10-6 secs/sec).
– High precision quartz clocks drift rate is about 10-7 or 10-8 secs/sec

Coordinated Universal Time (UTC)

• UTC is an international standard for time keeping
– It is based on atomic time, but occasionally adjusted to astronomical time
– International Atomic Time is based on very accurate physical clocks (drift rate 10-13)
• It is broadcast from radio stations on land and satellite (e.g.GPS)

• Computers with receivers can synchronize their clocks with these timing signals (by requesting time from GPS/UTC source)
– Signals from land-based stations are accurate to about 0.1-10 millisecond
– Signals from GPS are accurate to about 1 microsecond

11.3 Synchronizing physical clocks

Two models of synchronization
• External synchronization: a computer's clock *Ci* is synchronized with an external authoritative time source *S*, so that:
– $|S(t) - Ci(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of real time
– The clocks *Ci* are **accurate** to within the bound *D*.
• Internal synchronization: the clocks of a pair of computers are synchronized with one another so that:
– $| Ci(t) - Cj(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of real time
– The clocks *Ci* and *Cj* **agree** within the bound *D*.
Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively
– if the set of processes *P* is synchronized externally within a bound *D*, it is also internally synchronized within bound 2*D (worst case polarity)*

Clock correctness
• ***Correct clock***: a hardware clock *H* is said to be correct if its drift rate is within a bound $\rho > 0$ (e.g. 10-6 secs/ sec)
This means that the error in measuring the interval between real times *t* and *t'* is bounded:
– $(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$ (where $t' > t$)
– Which forbids jumps in time readings of hardware clocks

• ***Clock monotonicity***: weaker condition of correctness – $t' > t \Rightarrow C(t') > C(t)$
– e.g. required by Unix *make*
– A hardware clock that runs fast can achieve monotonicity by adjusting the values of α and β such that $Ci(t) = \alpha Hi(t) + \beta$
• ***Faulty clock***: a clock not keeping its correctness condition
– *crash failure* - a clock stops ticking
– *arbitrary* failure - any other failure
• e.g. jumps in time; Y2K bug

11.3.1 Synchronization in a synchronous system

A synchronous distributed system is one in which the following bounds are defined

– the time to execute each step of a process has known lower and upper bounds
– each message transmitted over a channel is received within a knownbounded time (min and max)
– each process has a local clock whose drift rate from real time has a known bound
 Internal synchronization in a synchronous system
– One process *p*1 sends its local time *t* to process *p*2 in a message *m*
– *p*2 could set its clock to *t* + *T*trans where *T*trans is the time to transmit *m*
– *T*trans is unknown but *min* ≤ *T*trans ≤ *max*
– uncertainty *u* = *max*-*min*. Set clock to *t* + (*max* - *min*)/2 then skew ≤ *u*/2

11.3.2 Cristian's method for an asynchronous system

• A time server *S* receives signals from a UTC source
– Process *p* requests time in *mr* and receives *t* in *mt* from *S*
– *p* sets its clock to *t* + *T*round/2
– Accuracy ± (*T*round/2 - *min*) :
• because the earliest time *S* puts *t* in message *mt* is *min* after *p* sent *mr*
• the latest time was *min* before *mt* arrived at *p*
• the time by *S*'s clock when *mt* arrives is in the range [*t*+*min*, *t* + *T*round - *min*]
• the width of the range is *T*round + 2*min*

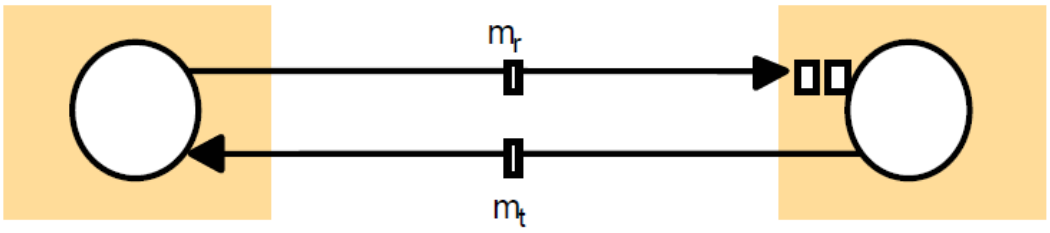


Figure 11.2

11.3.3 The Berkeley algorithm

• Problem with Cristian's algorithm
– a single time server might fail, so they suggest the use of a group of synchronized servers
– it does not deal with faulty servers
• Berkeley algorithm (also 1989)

– An algorithm for internal synchronization of a group of computers
– A *master* polls to collect clock values from the others (*slaves*)
– The master uses round trip times to estimate the slaves' clock values
– It takes an average (eliminating any above some average round trip time or with faulty clocks)
– It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
– Measurements
• 15 computers, clock synchronization 20-25 millisecs drift rate < 2x10-5
• If master fails, can elect a new master to take over (not in bounded time)

11.3.4 Network Time Protocol (NTP)

• A time service for the Internet - synchronizes clients to UTC
– Reliability from redundant paths, scalable, authenticates time sources
• Architecture
– Primary servers are connected to UTC sources
– Secondary servers are synchronized to primary servers
– Synchronization subnet - lowest level servers in users' computers
• strata: the hierarchy level

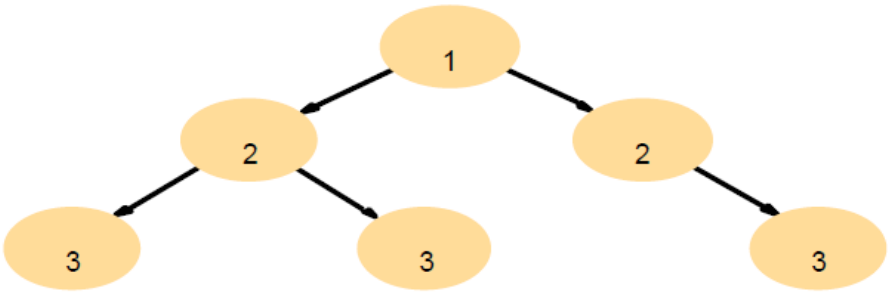


Figure 11.3 An example synchronization subnet in an NTP implementation

NTP - synchronization of servers

• The synchronization subnet can reconfigure if failures occur
– a primary that loses its UTC source can become a secondary
– a secondary that loses its primary can use another primary
• Modes of synchronization for NTP servers:
– Multicast
• A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
– Procedure call
• A server accepts requests from other computers (like Cristian's algorithm)
• Higher accuracy. Useful if no hardware multicast.
– Symmetric
• Pairs of servers exchange messages containing time information
• Used where very high accuracies are needed (e.g. for higher levels)

Messages exchanged between a pair of NTP peers

• All modes use UDP
• Each message bears timestamps of recent events:
– Local times of *Send* and *Receive* of previous message
– Local times of *Send* of current message
• Recipient notes the time of receipt *Ti* (we have Ti-3, Ti-2, Ti-1, Ti)
• In symmetric mode there can be a non-negligible delay between messages

Accuracy of NTP

• Estimations of clock offset and message delay
– For each pair of messages between two servers, NTP estimates an offset ***oi*** (between the two clocks) and a delay ***di*** (total time for the two messages, which take *t* and *t*')
Ti-2 = *Ti-3* + *t* + *o* and *Ti* = *Ti-1* + *t*' - *o*
– This gives us (by adding the equations) : *di* = *t* + *t*' = *T*i-2 - *Ti-3* + *T*i - *Ti-1*
– Also (by subtracting the equations)
o = *o*i + (*t*' - *t*)/2 where *oi* = (*T*i-2 - *Ti-3* + *Ti-1* - *Ti*)/2
– Using the fact that *t, t*'>0 it can be shown that
oi - *di* /2 ≤ *o* ≤ *oi* + *di* /2 .
• Thus *oi* is an estimate of the offset and *di* is a measure of the accuracy
• Data filtering
– NTP servers filter pairs <*oi, di*>, estimating reliability from variation (dispersions), allowing them to select peers; and synchronization based on the lowest dispersion or min *di* ok
• A relatively high filter dispersion represents relatively unreliable data
– Accuracy of tens of milliseconds over Internet paths (1 ms on LANs)

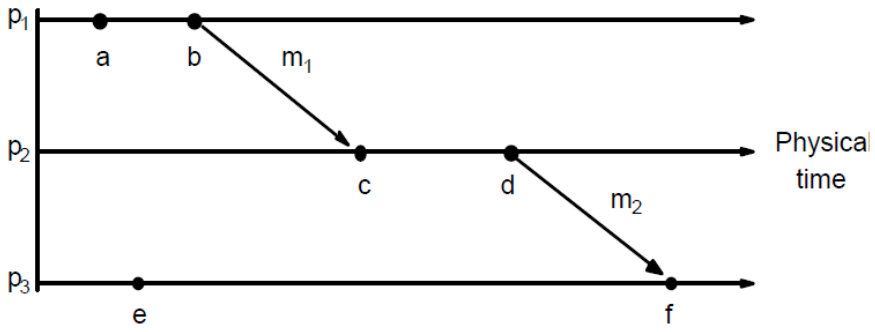
11.4 Logical time and logical clocks

• Instead of synchronizing clocks, event ordering can be used
1. If two events occurred at the same process *pi* ($i = 1, 2, \ldots N$) then theyoccurred in the order observed by *pi*, that is order □ $\rightarrow_i$
2. when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*

• Lamport[1978] generalized these two relationships into the

happened-before relation: $e \rightarrow_i e'$
– HB1: if $e \rightarrow_i e'$ in process *p*i, then $e \rightarrow e'$
– HB2: for any message *m*, send(*m*) → receive(*m*)
– HB3: if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Figure 11.5 Events occurring at three processes



• HB1: a→b, c→d, e→f
• HB2: b→c, d→f
• HB3: a→b→c→d→f
• a||e: a and e are concurrent (neither a→e nor e→a)

Lamport's logical clocks

• Each process *pi* has a logical clock ***Li***
– a monotonically increasing software counter
– not related to a physical clock
• Apply Lamport timestamps to events with happened-before relation
– LC1: *Li* is incremented by 1 before each event at process *pi*
– LC2:
(a) when process *pi* sends message *m*, it piggybacks $t = Li$
(b) when *pj* receives *(m,t)*, it sets Lj := *max*(Lj, *t*) and applies LC1 before timestamping the event *receive* (*m)*
• $e \rightarrow e'$ implies $L(e) < L(e')$, but $L(e) < L(e')$ does not imply $e \rightarrow e'$

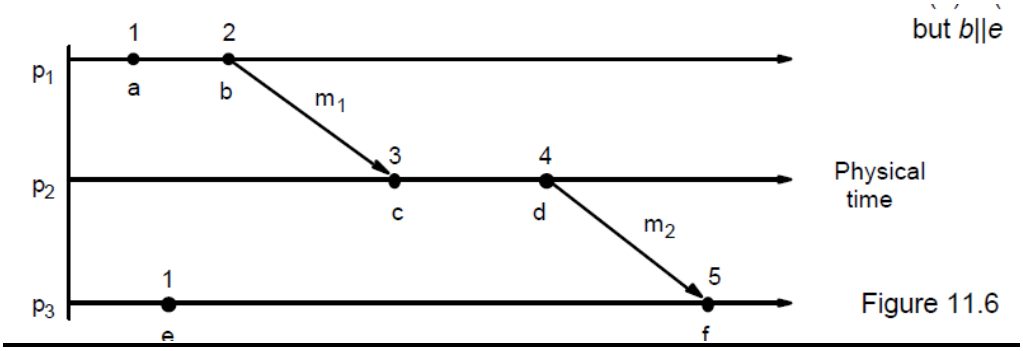


Figure 11.6

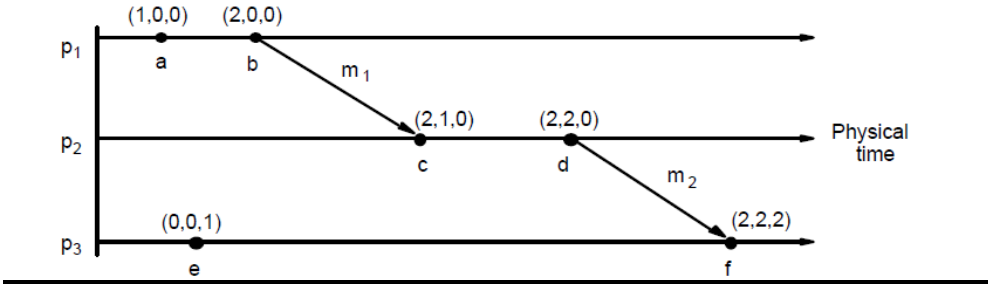
Totally ordered logical clocks

• Some pairs of distinct events, generated by different processes, may have numerically identical Lamport timestamps
– Different processes may have same Lamport time
• Totally ordered logical clocks
– If e is an event occurring at pi with local timestamp Ti, and if e' is an event occurring at pj with local timestamp Tj
– Define global logical timestamps for the events to be (*Ti, i*) and (*Tj, j*)
– Define (*Ti, i*) < (*Tj, j*) iff
• *Ti* < *Tj* or
• *Ti* = *Tj* and *i* < *j*
– No general physical significance since process identifiers are arbitrary

Vector clocks

• Shortcoming of Lamport clocks:
L(e) < *L(e')* doesn't imply $e \rightarrow e'$
• Vector clock: an array of N integers for a system of N processes
– Each process keeps its own vector clock *Vi* to timestamp local events
– Piggyback vector timestamps on messages
• Rules for updating vector clocks:
– *Vi*[*i*]] is the number of events that *pi* has timestamped
– *Vij*i] (*j*≠ *i*) is the number of events at *pj* that *pi* has been affected by
VC1: Initially, *Vi*[*j*] := 0 for *pi*, *j*=1.. *N* (*N* processes)
VC2: before *pi* timestamps an event, *Vi*[*i*] := *Vi*[*i*]+1
VC3: *pi* piggybacks *t* = *Vi* on every message it sends
VC4: when *pi* receives a timestamp *t*, it sets *Vi*[*j*] := max(*Vi*[*j*] , *t*[*j*]) for *j*=1..*N* (merge operation)

Figure 11.7 Vector timestamps for events shown in Figure 11.5

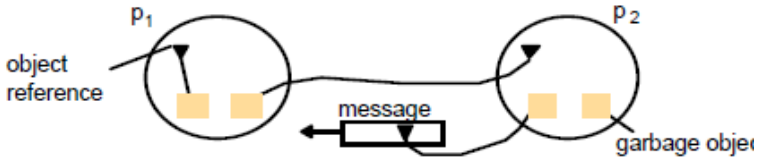


• Compare vector timestamps
– V=V' iff V[j] = V'[j] for j=1..N
– V>=V' iff V[j] <= V'[j] for j=1..N
– V<V' iff V<= V' ^ V!=V'
• Figure 11.7 shows
– *a*→*f* since V(a) < V(f)
– c || e since neither V(c) <= V(e) nor V(e) <= V(c)

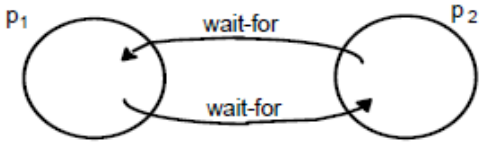
11.5 Global states

• How do we find out if a particular property is true in a distributed system? For examples, we will look at:
– Distributed Garbage Collection
– Deadlock Detection
– Termination Detection
– Debugging

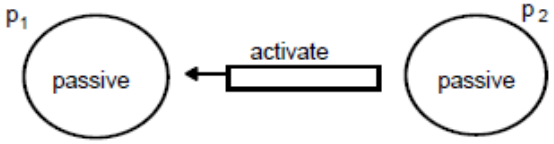
g



a. Garbage collection



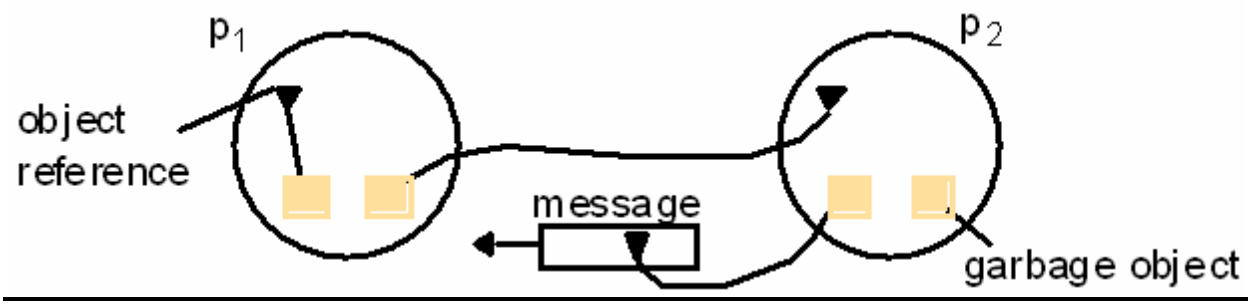
b. Deadlock



c. Termination

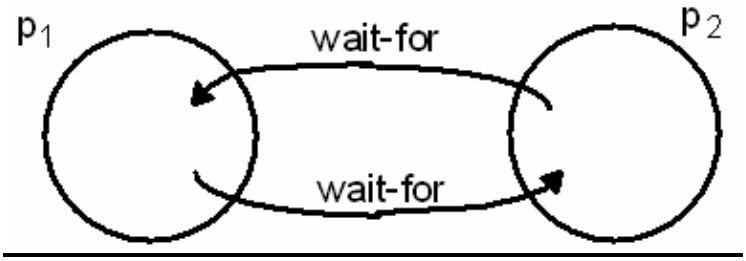
Distributed Garbage Collection

• Objects are identified as ***garbage*** when there are no longer any references to them in the system
• Garbage collection reclaims memory used by those objects
• In figure 11.8a, process p2 has two objects that do not have any references to other objects, but one object does have a reference to a message in transit. It is not garbage, but the other p2 object is
• Thus we must consider communication channels as well as object references to determine unreferenced objects



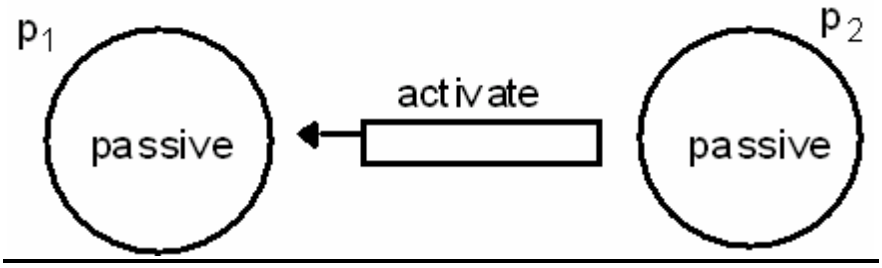
Deadlock Detection

• A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and there is a cycle in the graph of the ***waits-for*** relationship
• In figure 11.8b, both p1 and p2 wait for a message from the other, so both are blocked and the system cannot continue



Termination Detection

• It is difficult to tell whether a distributed algorithm has terminated. It is not enough to detect whether each process has halted
• In figure 11.8c, both processes are in passive mode, but there is an activation request in the network
• Termination detection examines multiple states like deadlock detection, except that a deadlock may affect only a portion of the processes involved, while *termination detection must ensure that all of the processes have completed*


p1
p2
activate
passive
passive

Distributed Debugging

• Distributed processes are complex to debug. One of many possible problems is that consistency restraints must be evaluated for simultaneous attribute values in multiple processes at different instants of time.

• All four of the distributed problems discussed in this section have particular solutions, but all of them also illustrate the need to observe global states. We will now look at a general approach to observing global states.

• Without global time identified by perfectly synchronized clocks, the ability to identify successive states in an individual process does not translate into the ability to identify successive states in distributed processes

• We can assemble meaningful global states from local states recorded at different local times in many circumstances, but must do so carefully and recognize limits to our capabilities

• A general system P of N processes pi (i=1..N)

– pi's history: history(pi)=hi=<ei0, ei1, ei2, …>

– finite prefix of pi's history: hi

k= <ei0, ei1, ei2, …, eik>

– state of pi immediately before the kth event occurs: sik

– global history H=h1 U h2 U…U hN

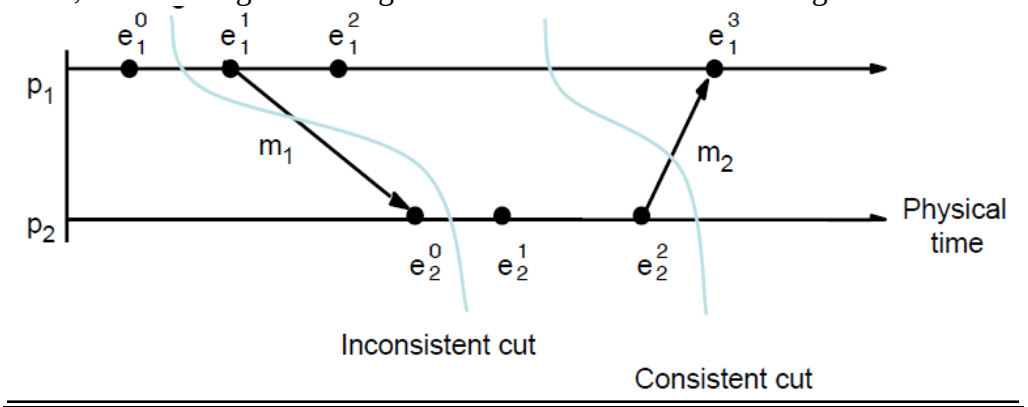
– A cut of the system's execution is a subset of its global history that is a union of prefix of process histories C=h1c1 U h2c2 U…U hNcN

The following figure gives an example of an inconsistent cutic and a consistent cutcc. The distinguishing characteristic is that

– cutic includes the receipt of message m1 but *not* the sending of it, while

– cutcc includes the sending *and* receiving of m1 *and* cuts between the sending and receipt of the message m2.

• A consistent cut cannot violate temporal causality by implying that a result occurred before its cause, as in message m1 being received before the cut and being sent after the cut.



11.5.2 Global state predicates

• A Global State Predicate is a function that maps from the set of global process states to True or False.
• Detecting a condition like deadlock or termination requires evaluating a Global State Predicate.
• A Global State Predicate is stable: once a system enters a state where it is true, such as deadlock or termination, it remains true in all future states reachable from that state.
However, when we monitor or debug an application, we are interested in non stable predicates.

11.5.3 The Snapshot Algorithm

• Chandy and Lamport defined a snapshot algorithm to determine global states of distributed systems
• The goal of a snapshot is to record a set of process and channel states (a snapshot) for a set of processes so that, even if the combination of recorded states may not have occurred at the same time, the recorded global state is consistent
– The algorithm records states locally; it does not gather global states at one site.
• The snapshot algorithm has some assumptions
– Neither channels nor processes fail
– Reliable communications ensure every message sent is received exactly once
– Channels are unidirectional
– Messages are received in FIFO order
– There is a path between any two processes
– Any process may initiate a global snapshot at any time
– Processes may continue to function normally during a snapshot

Snapshot Algorithm

• For each process, incoming channels are those which other processes can use to send it messages. Outgoing channels are those it uses to send messages. Each process records its state and for each incoming channel a set of messages sent to it. The process records for each channel, any messages sent after it recorded its state and before the sender recorded its own state. This approach can differentiate between states in terms of messages transmitted but not yet received
• The algorithm uses special marker messages, separate from other messages, which prompt the receiver to save its own state if it has not done so and which can be used to determine which messages to include in the channel state.
• The algorithm is determined by two rules

Figure 11.10 Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process pi On *pi*'s receipt of a *marker* message over channel *c*:
if (*pi* has not yet recorded its state) it records its process state now; records the state of *c* as the empty set; turns on recording of messages arriving over other incoming channels;
else
pi records the state of *c* as the set of messages it has received over *c* since it saved its state.
end if
Marker sending rule for process pi
After *pi* has recorded its state, for each outgoing channel *c*: *pi* sends one marker message over *c* (before it sends any other message over *c*).

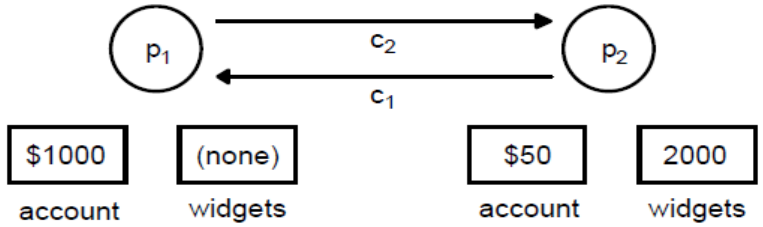
Example

•Figure 11.11 shows an initial state for two processes.

•Figure 11.12 shows four successive states reached and identified after state transitions by the two processes.

•Termination: it is assumed that all processes will have recorded their states and channel states a finite time after some process initially records its state.

Figure 11.11 Two processes and their initial states



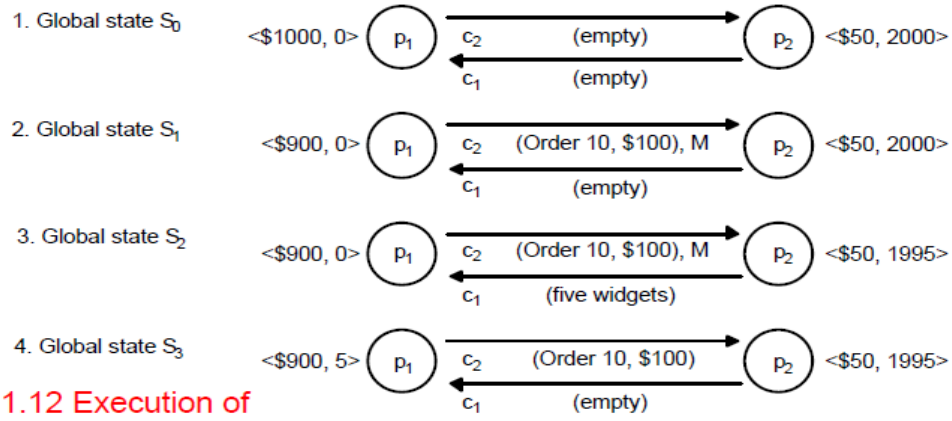


Figure 11.12 Execution of processes in Figure 11.11

Characterizing a state

• A snapshot selects a consistent cut from the history of the execution. Therefore the state recorded is consistent. This can be used in an ordering to include or exclude states that have or have not recorded their state before the cut. This allows us to distinguish events as pre-snap or post-snap events.

• The reachability of a state (figure 11.13) can be used to determine stable predicates.

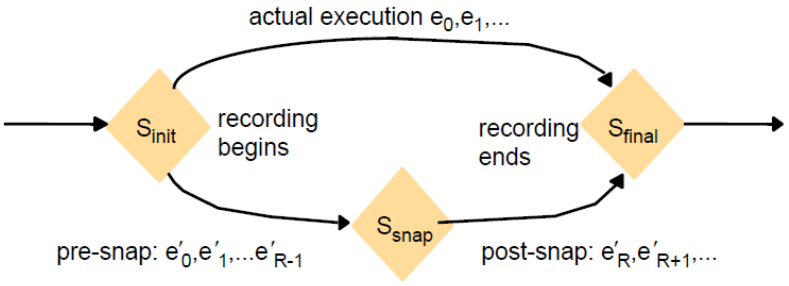


Figure 11.13 Reachability between states in the snapshot algorithm

Introduction

• Fundamental issue: for a set of processes, how to coordinate their actions or to agree on one or more values?
– even no fixed master-slave relationship between the components
• Further issue: how to consider and deal with failures when designing algorithms
• Topics covered
– mutual exclusion
– how to elect one of a collection of processes to perform a special role
– multicast communication
– agreement problem: consensus and byzantine agreement

Failure Assumptions and Failure Detectors

• Failure assumptions of this chapter
– Reliable communication channels
– Processes only fail by crashing unless state otherwise
• Failure detector: object/code in a process that detects failures of other processes
• unreliable failure detector
– One of two values: unsuspected or suspected
• Evidence of possible failures
– Example: most practical systems
• Each process sends “alive/I’m here” message to everyone else
• If not receiving “alive” message after timeout, it’s suspected
– maybe function correctly, but network partitioned
• reliable failure detector
– One of two accurate values: unsuspected or failure – few practical systems

12.2 Distributed Mutual Exclusion

• Process coordination in a multitasking OS
– **Race condition**: several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place
– **critical section**: when one process is executing in a critical section, no other process is to be allowed to execute in its critical section
– **Mutual exclusion**: If a process is executing in its critical section, then no other processes can be executing in their critical sections
• Distributed mutual exclusion
– Provide critical region in a distributed environment
– message passing
for example, locking files, locked daemon in UNIX (NFS is stateless, no file-locking at the NFS level)

Algorithms for mutual exclusion

• Problem: an asynchronous system of *N* processes
– processes don't fail
– message delivery is reliable; not share variables
– only one critical region
– application-level protocol: enter(), resourceAccesses(), exit()
• Requirements for mutual exclusion
– Essential
• [ME1] safety: only one process at a time
• [ME2] liveness: eventually enter or exit
– Additional
• [ME3] happened-before ordering: ordering of enter() is the same as HB ordering
• Performance evaluation
– overhead and bandwidth consumption: # of messages sent
– client delay incurred by a process at entry and exit
– throughput measured by synchronization delay: delay between one's exit and next's entry

A central server algorithm

• server keeps track of a token---permission to enter critical region
– a process requests the server for the token
– the server grants the token if it has the token
– a process can enter if it gets the token, otherwise waits
– when done, a process sends release and exits

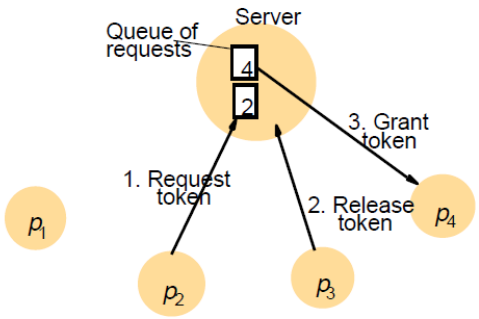


Figure 12.2 Server managing a mutual exclusion token for a set of processes

A central server algorithm: discussion

• Properties
– safety, why?
– liveness, why?
– HB ordering not guaranteed, why?
• Performance
– enter overhead: two messages (request and grant)
– enter delay: time between request and grant
– exit overhead: one message (release)
– exit delay: none
– synchronization delay: between release and grant
– centralized server is the bottleneck

A ring-based algorithm

• Arrange processes in a logical ring to rotate a token
– Wait for the token if it requires to enter the critical section
– The ring could be unrelated to the physical configuration
• *pi* sends messages to $p(i+1) \bmod N$
– when a process requires to enter the critical section, waits for the token
– when a process holds the token
• If it requires to enter the critical section, it can enter
– when a process releases a token (exit), it sends to its neighbor
• If it doesn't, just immediately forwards the token to its neighbor

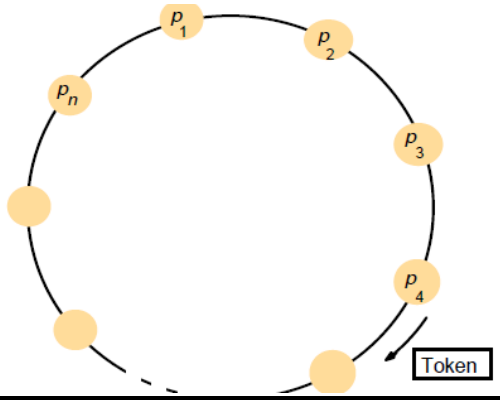


Figure 12.3 A ring of processes transferring a mutual exclusion token

An algorithm using multicast and logical clocks

• Multicast a request message for the token (Ricart and Agrawala [1981])
– enter only if all the other processes reply
– totally-ordered timestamps: <*T*, *pi* >
• Each process keeps a *state: RELEASED*, *HELD*, *WANTED*
– if all have *state* = *RELEASED*, all reply, a process can hold the token and enter
– if a process has *state* = *HELD*, doesn't reply until it exits
– if more than one process has *state* = *WANTED*, process with the lowest timestamp will get all *N*-1 replies first

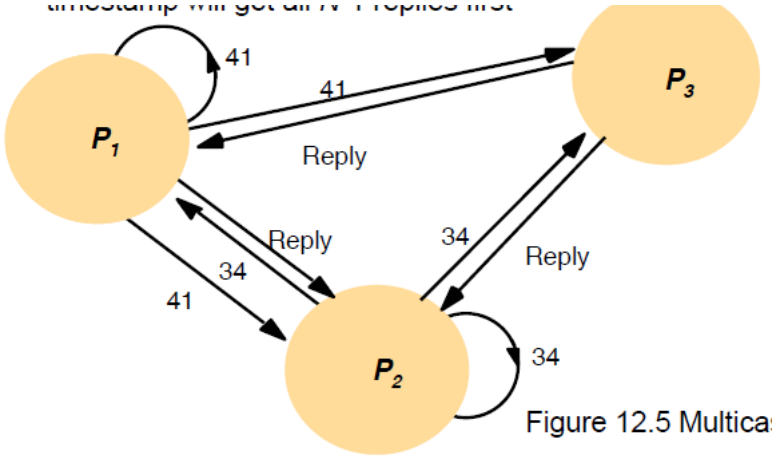


Figure 12.5 Multicast synchronization

Figure 12.4 Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

An algorithm using multicast: discussion

•Properties
– safety, why?
– liveness, why?
– HB ordering, why?
• Performance
– bandwidth consumption: no token keeps circulating
– entry overhead: 2(*N*-1), why? [with multicast support: 1 + (*N* -1) = *N*]
– entry delay: delay between request and getting all replies
– exit overhead: 0 to *N*-1 messages
– exit delay: none
– synchronization delay: delay for 1 message (one last reply from the previous holder)

Maekawa's voting algorithm

•Observation: not all peers to grant it access

– Only obtain permission from subsets, overlapped by any two processes

•Maekawa's approach

– subsets Vi,Vj for process Pi, Pj

• Pi ∈ Vi, Pj ∈ Vj

• Vi ∩ Vj ≠ ∅, there is at least one common member

• subset |Vi|=K, to be fair, each process should have the same size

– Pi cannot enter the critical section until it has received all K reply messages

– Choose a subset

• Simple way (2√N): place processes in a √N by √N matrix and let Vi be the union of the row and column containing Pi

• Optimal (√N): non-trivial to calculate (skim here)

– Deadlock-prone

• V1={P1, P2}, V2={P2, P3}, V3={P3, P1}

• If P1, P2 and P3 concurrently request entry to the critical section, then its possible that each process has received one (itself) out of two replies, and none can proceed

• adapted and solved by [Saunders 1987]

Figure 12.6 Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;

For p_i *to enter the critical section*
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;

On receipt of a request from p_i *at* p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i *to exit the critical section*
  state := RELEASED;
  Multicast *release* to all processes in V_i;

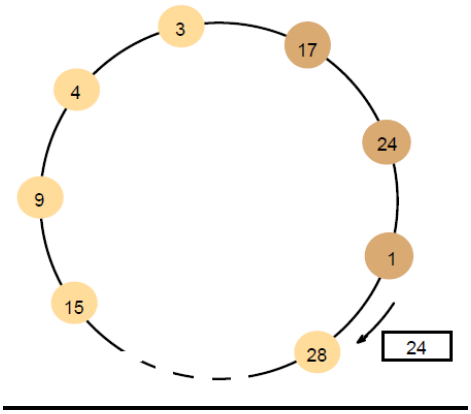
On receipt of a release from p_i *at* p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Elections

• Election: choosing a unique process for a particular role
  – All the processes agree on the ***unique*** choice
  – For example, server in dist. mutex
  • Assumptions
  – Each process can call only one election at a time
  – multiple concurrent elections can be called by different processes
  – Participant: engages in an election
  • each process *pi* has variable *electedi* = ? (don't know) initially
  • process with the *largest* identifier wins
  – The (unique) identifier could be any useful value
  • Properties
  – [E1] *electedi* of a "participant" process must be P (elected process=largestid) or ⊥ (undefined)
  – [E2] liveness: all processes participate and eventually set *electedi* != ⊥ (or crash)
  • Performance
  – overhead (bandwidth consumption): # of messages
  – turnaround time: # of messages to complete an election

A ring-based election algorithm
  • Arrange processes in a logical ring
  – *pi* sends messages to *p*(*i*+1) mod *N*
  – It could be unrelated to the physical configuration
  – Elect the coordinator with the largest id
  – Assume no failures
  • Initially, every process is a non-participant. Any process can call an election
  – Marks itself as participant
  – Places its id in an *election* message
  – Sends the message to its neighbor
  – Receiving an election message
  • if *id* > *myid*, forward the msg, mark participant
  • if *id* < *myid*
  – non-participant: replace *id* with *myid*: forward the msg, mark participant
  – participant: stop forwarding (why? Later, multiple elections)
  • if *id* = *myid*, coordinator found, mark non-participant, *electedi* := *id*, send *elected* message with *myid*
  – Receiving an elected message
  • *id* != *myid*, mark non-participant, *electedi* := *id* forward the msg
  • if *id* = *myid*, stop forwarding

Figure 12.7 A ring-based election in progress



• Receiving an election message:
– if *id* > *myid*, forward the msg, mark participant
– if *id* < *myid*
• non-participant: replace *id* with *myid*: forward the msg, mark participant
• participant: stop forwarding (why? Later, multiple elections)
– if *id* = *myid*, coordinator found, mark non-participant, *electedi* := *id*, send *elected* message with *myid*
• Receiving an elected message: – *id* != *myid*, mark non-participant,
electedi := *id* forward the msg
– if *id* = *myid*, stop forwarding

A ring-based election algorithm: discussion

•Properties
– safety: only the process with the largest id can send an *elected* message
– liveness: every process in the ring eventually participates in the election; extra elections are stopped
• Performance
– one election, best case, when?
• *N election* messages
• *N elected* messages
• turnaround: 2*N* messages
– one election, worst case, when?
• 2*N* - 1 election messages
• *N elected* messages
• turnaround: 3*N* - 1 messages
– can't tolerate failures, not very practical

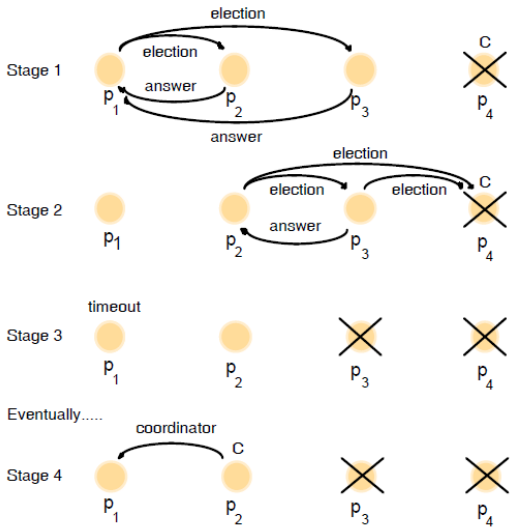
The bully election algorithm

•Assumption

– Each process knows which processes have higher identifiers, and that it can communicate with all such processes
•Compare with ring-based election
– Processes can crash and be detected by timeouts
• synchronous
• timeout $T = 2T_{transmitting}$ (max transmission delay) + $T_{processing}$ (max processing delay)
•Three types of messages
– Election: announce an election
– Answer: in response to Election
– Coordinator: announce the identity of the elected process

The bully election algorithm: howto
• Start an election when detect the coordinator has failed or begin to replace the coordinator, which has lower identifier
– Send an election message to all processes with higher id's and waits for answers (except the failed coordinator/process)
• If no answers in time *T*
– Considers it is the coordinator
– sends coordinator message (with its id) to all processes with lower id's
• else
– waits for a coordinator message and starts an election if T' timeout
– To be a coordinator, it has to start an election
• A higher id process can replace the current coordinator (hence "bully")
– The highest one directly sends a coordinator message to all process with lower identifiers
• Receiving an election message
– sends an answer message back
– starts an election if it hasn't started one—send election messages to all higher-id processes (including the "failed" coordinator—the coordinator might be up by now)
• Receiving a coordinator message
– set *electedi* to the new coordinator

Figure 12.8 The bully algorithm



The election of coordinator p_2, after the failure of p_4 and then p_3

The bully election algorithm: discussion

• Properties

– safety:

• a lower-id process always yields to a higher-id process

• However, it's guaranteed

– if processes that have crashed are replaced by processes with the same identifier since message delivery order might not be guaranteed and

– failure detection might be unreliable

– liveness: all processes participate and know the coordinator at the end

• Performance

– best case: when?

• overhead: *N*-2 *coordinator* messages

• turnaround delay: no *election/answer* messages

– worst case: when?

• overhead:

• 1+ 2 + ...+ (*N*-2) + (*N*-2)= (*N*-1)(*N*-2)/2 + (*N*-2) *election* messages,

• 1+...+ (*N*-2) *answer* messages,

• *N*-2 *coordinator* messages,

• total: (*N*-1)(*N*-2) + 2(*N*-2) = (*N*+1)(*N*-2) = O(*N*2)

– turnaround delay: delay of election and answer messages

Multicast Communication

• Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
– The set of messages that every process of the group should receive
– On the delivery ordering across the group members
• Challenges
– Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
– Delivery guarantees ensure that operations are completed
• Types of group
– Static or dynamic: whether joining or leaving is considered
– Closed or open
• A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
• A group is open if processes outside the group can send to it

Reliable Multicast

• Simple basic multicasting (B-multicast) is sending a message to every process that is a member of a defined group
– B-multicast (g, m) for each process p ∈ group g, send (p, message m)
– On receive (m) at p: B-deliver (m) at p
• Reliable multicasting (R-multicast) requires these properties
– Integrity: a correct process sends a message to only a member of the group and does it only once
– Validity: if a correct process sends a message, it will eventually be delivered
– Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

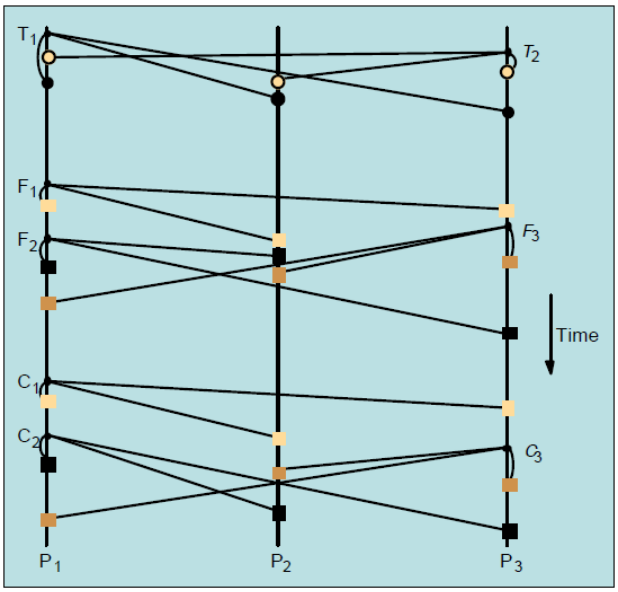
Figure 12.10 Reliable multicast algorithm

```
On initialization
    Received := {};

For process p to R-multicast message m to group g
    B-multicast(g, m);          // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)
    if (m ∉ Received)
    then
                Received := Received ∪ {m};
                if (q ≠ p) then B-multicast(g, m); end if
                R-deliver m;
    end if
```

Implementing reliable R-multicast over B-multicast
– When a message is delivered, the receiving process multicasts it
– Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering
•Three types of message ordering
– ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other
– ***Casual ordering***: any correct process that delivers the second message will deliver the previous message first
– ***Total ordering***: if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first
•Note that
– FIFO ordering and casual ordering are only partial orders
– Not all messages are sent by the same sending process
– Some multicasts are concurrent, not able to be ordered by happened before
– Total order demands consistency, but not a particular order

Figure 12.12 Total, FIFO and causal ordering of multicast messages



Notice
• the consistent ordering of totally ordered messages *T1* and *T2*,
• the FIFO-related messages *F1* and *F2* and
• the causally related messages *C1* and *C3* and
• the otherwise arbitrary delivery ordering of messages

Note that *T1* and *T2* are delivered in opposite order to the physical time of message creation

Bulletin board example (FIFO ordering)

• A bulletin board such as Web Board at NJIT illustrates the desirability of consistency and FIFO ordering. A user can best refer to preceding messages if they are delivered in order. Message 25 in Figure 12.13 refers to message 24, and message 27 refers to message 23.

• Note the further advantage that Web Board allows by permitting messages to begin threads by replying to a particular message. Thus messages do not have to be displayed in the same order they are delivered

Bulletin board: *os.interesting*

Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 12.13 Display from bulletin board program

Implementing total ordering

• The normal approach to total ordering is to assign totally ordered identifiers to multicast messages, using the identifiers to make ordering decisions.
• One possible implementation is to use a sequencer process to assign identifiers. See Figure 12.14. A drawback of this is that the sequencer can become a bottleneck.
• An alternative is to have the processes collectively agree on identifiers. A simple algorithm is shown in Figure 12.15.

Figure 12.14 Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
  B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);

On B-deliver(<*m, i*>) *with* $g = group(m)$
  Place <*m, i*> in hold-back queue;

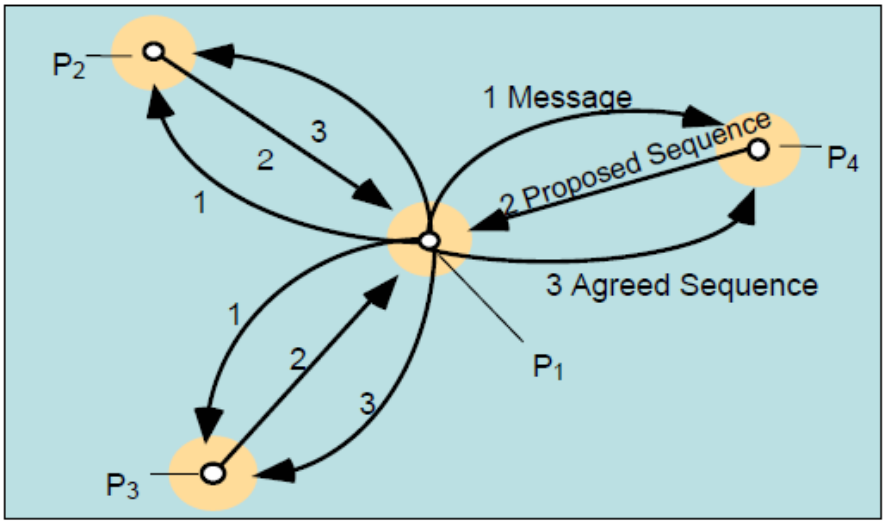
On B-deliver(m_{order} = <"order", *i, S*>) *with* $g = group(m_{order})$
  wait until <*m, i*> in hold-back queue and $S = r_g$;
  TO-deliver m; // (after deleting it from the hold-back queue)
  $r_g = S + 1$;

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with* $g = group(m)$
  B-multicast(*g*, <"order", *i*, s_g>);
  $s_g := s_g + 1$;

Figure 12.15 The ISIS algorithm for total ordering



Each process q in group g keeps
• Aq g: the largest agreed sequence number it has observed so far for the group g
• Pq g: its own largest proposed sequence number

Algorithm for process p to multicast a message m to group g
1. B-multicasts <m, i> to g, where i is a unique identifier for m
2. Each process q replies to the sender p with a proposal for the message's agreed sequence number of Pq g :=Max(Aq g, Pq g)+1
3. Collects all the proposed sequence numbers and selects the largest one a as the next agreed sequence number. It then B-multicasts <i, a> to g.

4. Each process q in g sets Aq g := Max(Aq g, a) and attaches a to the message identified by i

Implementing casual ordering

• Causal ordering using vector timestamps (Figure 12.16)
– Only orders multicasts, and ignores one-to-one messages between processes
– Each process updates its vector timestamp before delivering a message to maintain the count of precedent messages

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization
$V_i^g[j] := 0$ $(j = 1, 2\ldots, N)$;

To CO-multicast message m to group g
$V_i^g[i] := V_i^g[i] + 1$;
B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$
place $<V_j^g, m>$ in hold-back queue;
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;
CO-deliver m; // after removing it from the hold-back queue
$V_i^g[j] := V_i^g[j] + 1$;

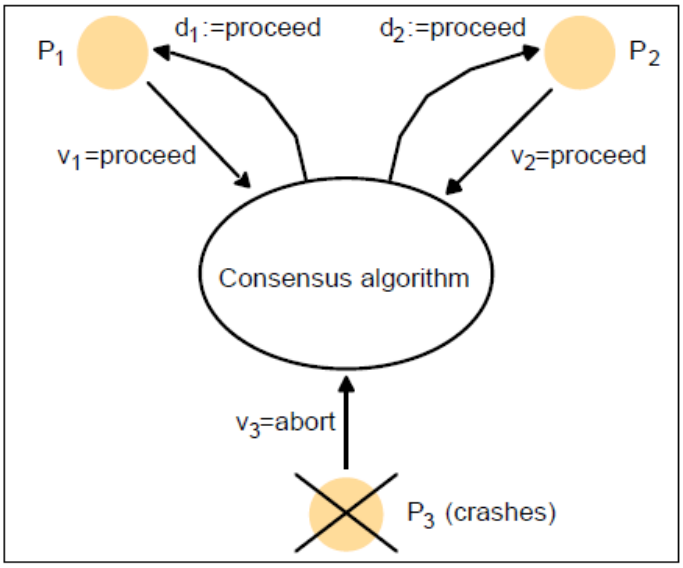
Lecture Notes

Consensus and related problems

• Problems of agreement
– For processes to agree on a value (consensus) after one or more of the processes has proposed what that value should be
– Covered topics: byzantine generals, interactive consistency, totally ordered multicast
• The byzantine generals problem: a decision whether multiple armies should attack or retreat, assuming that united action will be more successful than some attacking and some retreating
• Another example might be space ship controllers deciding whether to proceed or abort. Failure handling during consensus is a key concern
• Assumptions
– communication (by message passing) is reliable
– processes may fail
• Sometimes up to f of the N processes are faulty

Consensus Process

1. Each process pi begins in an undecided state and proposes a single value vi, drawn from a set D (i=1…N)
2. Processes communicate with each other, exchanging values
3. Each process then sets the value of a decision variable di and enters the decided state



Two processes propose "proceed." One proposes "abort," but then crashes. The two remaining processes decide proceed.

Figure 12.17 Consensus for three processes

Requirements for Consensus

• Three requirements of a consensus algorithm
– ***Termination***: Eventually every correct process sets its decision variable
– ***Agreement***: The decision value of all correct processes is the same: if pi and pj are correct and have entered the *decided* state, then di=dj
(i,j=1,2, …, N)
– ***Integrity***: If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value

The byzantine generals problem

• Problem description
– Three or more generals must agree to *attack* or to *retreat*
– One general, the *commander*, issues the order
– Other generals, the *lieutenants*, must decide to attack or retreat
– One or more generals may be treacherous
• A *treacherous general* tells one general to attack and another to retreat
• Difference from consensus is that a single process supplies the value to agree on
• Requirements
– *Termination*: eventually each correct process sets its decision variable
– *Agreement*: the decision variable of all correct processes is the same
– *Integrity*: if the commander is correct, then all correct processes agree on the value that the commander has proposed (but the commander need not be correct)

The interactive consistency problem

• Interactive consistency: all correct processes agree on a vector of values, one for each process. This is called the decision vector
– Another variant of consensus
• Requirements
– *Termination*: eventually each correct process sets its decision variable
– *Agreement*: the decision vector of all correct processes is the same
– *Integrity*: if any process is correct, then all correct processes decide the correct value for that process

Relating consensus to other problems

• Consensus (C), Byzantine Generals (BG), and Interactive Consensus (IC) are all problems concerned with making decisions in the context of arbitrary or crash failures
• We can sometimes generate solutions for one problem in terms of another. For example
– We can derive IC from BG by running BG N times, once for each process with that process acting as commander
– We can derive C from IC by running IC to produce a vector of values at each process, then applying a function to the vector's values to derive a single value.
– We can derive BG from C by
• Commander sends proposed value to itself and each remaining process
• All processes run C with received values
• They derive BG from the vector of C values

Consensus in a Synchronous System

• Up to f processes may have crash failures, all failures occurring during f+1 rounds. During each round, each of the correct processes multicasts the values among themselves
• The algorithm guarantees all surviving correct processes are in a position to agree
• Note: any process with f failures will require at least f+1 rounds to agree

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization
$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f + 1)$
B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent
$Values_i^{r+1} := Values_i^r$;
while (in round *r*)
{
On B-deliver(V_j) *from some* p_j
$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;
}

After $(f + 1)$ *rounds*
Assign $d_i = minimum(Values_i^{f+1})$;

Limits for solutions to Byzantine Generals

• Some cases of the Byzantine Generals problems have no solutions
– Lamport *et al* found that if there are only 3 processes, there is no solution
– Pease *et al* found that if the total number of processes is less than three times the number of failures plus one, there is no solution
• Thus there is a solution with 4 processes and 1 failure, if there are two rounds
– In the first, the commander sends the values
– while in the second, each lieutenant sends the values it received

Figure 12.19 Three Byzantine generals

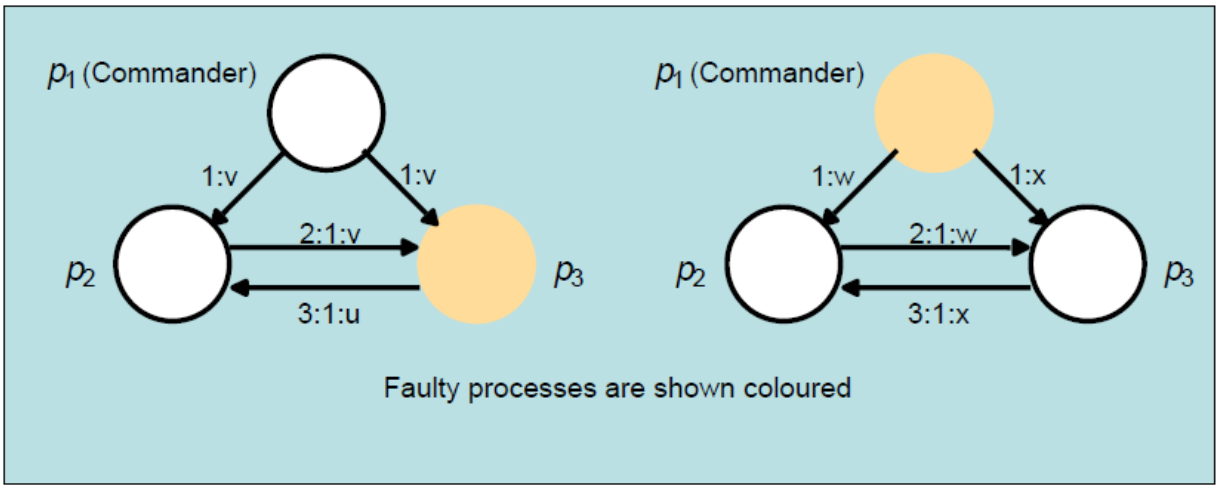
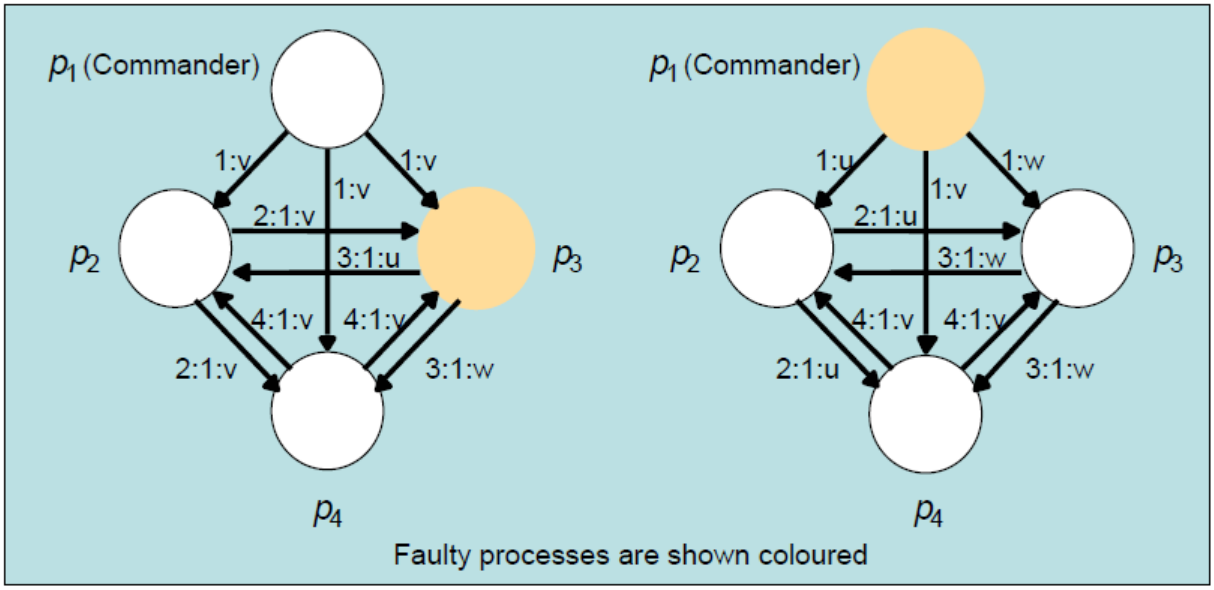


Figure 12.20 Four Byzantine generals



Asynchronous Systems

• All solutions to consistency and Byzantine generals problems are limited to synchronous systems

• Fischer *et al* found that there are no solutions in an asynchronous system with even one failure

• This impossibility is circumvented by *masking faults* or using *failure detection*

• There is also a partial solution, assuming an *adversary* process, based on *introducing random values* in the process to prevent an effective thwarting strategy. This does not always reach consensus