**UNIT – 4**

### 4.1 TIME AND GLOBAL STATES - INTRODUCTION

* Time is an important and interesting issue in distributed system, *for* several rear-First, time is a quantity we often want to measure accurately.

* In order to know at what time particular event occurred at a particular computer, it is necessary to synchronize i t s clock with an authoritative external source of time.

* Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution.

* These include maintain the consistency of distributed data *checking the* authenticity of a request service a server and eliminating the processing of duplicate updates.

* The relative order of two event can be reversed for two different observers.

* But t h i s cannot happen if one event could have caused the other to occur.

* In that case, the physical effect follows t h e physical cause for observers, although the time elapsed between cause and effect can vary. The timing of physical events is relative to the observer.

* There is no special physical clock in the universe to which we can appeal when we want to measure intervals of time.

* The notion of physical time is also problematic in a distributed system.

* This is not due to effects of special relativity, which are negligible or non-existent for normal computers.

* The problem is based on a similar limitation in our ability to timestamp events at different nodes sufficiently accurately to know the order in which any pair of events occurred, or whether they occurred simultaneously.

* There is no absolute, global time that we can appeal to.

* Yet we sometimes need to observe distributed systems and establish whether certain states of affairs occurred at the same time.

* The absence of global physical time makes it difficult to find out the state of our distributed programs as they execute.

* Let us consider how to order and timestamp the events that occur at a single process.

* Consider a distributed system consisting of a collection of N processes Pi i =1, 2, 3 . . . N.

* Each process executes on a single processor, and the processors do not share memory.

* Each process P i has state *si* which in general, transforms as it executes.

* The process's state includes the values if a l l the variables within it.

* The state may also include the values of any objects in its local operating system environment that it affects, such as files.

* As each process Pi executes, it takes a series of actions, each of which is either a message sends or receives operation, or an operation that transforms Pi‘s state.

* We define an event to be the occurrence of a single a ct i o n that a process carries out as if executes - a communication action or a state-transforming section.

* The sequence of events within a single process Pi can be placed in a single, total ordering, using the relation

—>ibetween the events.

* That is. ei *—>i* ei if and only if the event e occurs before e‘ at Pi.

* The *history* of a process Pi is defined as a series of events that take place within it, ordered by the relation

history (Pi ) = h = <ei0, ei , ei . >

###  Clocks

* Each computer contains their own physical clock.

* These clocks are electronic devices that counts oscillations occurring in a crystal al a definite frequency, and that typically divide this count and store the result in a counter register.

* The operating system reads the node's hardware clock value H2(t), scales it and adds an offset so as to produce a software clock Ci*( t ) =άH! ( t ) —>* β that approximately measures real, physical time t for process Pi

* We can use i t s value to timestamp any event at Pi

###  Clock skew and clock drift

* Computer clocks will not be in perfect agreement.

* The instantaneous difference between the readings of any two clocks is called ***clock skew****.*

* Also, the crystal-based clocks may be subject to *clock drift,* which means that they count time at different rates, and so diverge.

* The underlying oscillators are subject to physical variations with the consequence that their frequencies of oscillation differ.

* A clock's drift rate is the range in the offset between the clock and a nominal perfect reference clock per unit of time measured by the reference clock.

* For ordinary clocks based on a quartz crystal, this is about '' seconds/second - giving a difference of l sec every 10, 00,000 seconds or 11.6 days.

###  Coordinated universal time

* If is abbreviated as UTC. It is an international standard for timekeeping.

* It is based on atomic time.

* UTC signals are synchronized and broadcast regularly from land based radio stations and satellites covering many parts of the world.

* Computers with receivers attached can synchronize their clocks with these timing signals.

* Computers may also receive the time to an accuracy of a few milliseconds over a telephone line.

**4.2 SYNCHRONIZING PHYSICAL CLOCKS.**

* In order to know at what time of day events occur at the processes in a distributed system, it is necessary to synchronize the processes' clock Ci with an Ci authoritative, external source of time, this is **external synchronization.**

* If the clocks are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks - even though they are not synchronized externally.This is **internal synchronization**.

Two modes of synchronization can be defined more closely as follows, over an interval of real time I.

* + - * **External synchronization:** For a synchronization bound D>0. and for a source S of **UTC** time,

|*s(t) - C*i*(t)| < D ,* for I = 1, 2 . . . . N and for all real times t in I.

* + - * **Internal synchronization**: for a synchronization bound D>0. |Ci (t) – Cj ( t ) for i, j = 1,

2, 3. . . N and for all times t in I.

The algorithms used for external and internal synchronizations are as follows:

**4.2.1 Synchronization in a synchronous system.**

* In a synchronous system, bounds are known for the drift rate of clocks, the maximum message transmission delay and the time for execution.

* One process sends the time t on its local clock to the other in a message m.

* The receiving process could set i t s clock to the time *t* + *Ttrans* where *Ttrans* is the time taken the transmit m between them.

* Unfortunately, *Ttrans* is subject to variation and is unknown.

* In a synchronous system, by definition, there is a lower bound (min) and an upper bound (max) on the time taken to transmit any message.

* Let the uncertainty in the message transmission time be u, so that u= (max-min).

* If i t s receiver sets its clock to be t+min, then the clock skew may be as much as u, since the message may in fact have taken time max to arrive.

* Similarly, if it sets its clock to t+max, the skew may again be as large as u. If however, it sets its clock to the half-way peit, t+(max+min)/2, then the skew is at most u/2.

* In general, the optimum bound that can be achieved on clock skew when synchronizing N clocks is u(l - 1/N).

**4.2.2 Cristian's method for synchronizing clocks:**

* Cristian suggested the use of a time server, connected to a device that receives signals from a source UTC, to synchronize computers externally.

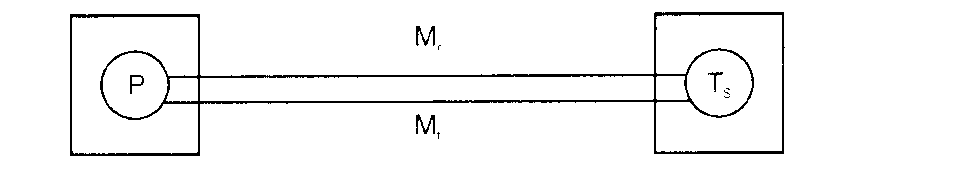
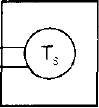
* Upon request, the server process S supplies the time according to i t s clocks, as shown in figure 3.13.

* A process p requests the time in a message mr and receives the time value t in the message mi.

* Process p records the total round trip-time Tround taken to send the request mr and receive the reply mi A simple estimate of the time to which p should set its clock is t+Tround/2, which assumes that the elapsed time is split equally before and after S placed t in mi.

* If the value of the minimum transmission time min is known or can be conservatively estimated, then we can determine the accuracy as follows:

* + - * The earliest point at which S could have placed the time in mi was min after p dispatched mr.
      * The latest point at which it could have done this was 'min' before mt arrived at p.
      * The time by S 's clock whe n the repl y messa ge ar ri ves is therefore in the range [t + min, t+ Tround- min] the width of this range is -2 min, so the accuracy is ± (Tround/*2 -* min).



***Figure 3.13. Synchronization using a time serve****r*

#### 4.2.3 The Berkeley algorithm

* In Berkeley algorithm, a coordinator computer is chosen to act as the master. This computer periodically polls the other computers whose clocks are to be synchronized, called slaves.

* The slaves send back their clock values to it.

* The master estimates their local clock times by observing the round-trip times and it averages the values obtained including its own clock's reading.

* Instead of sending the update current time back to the other computers, the master sends the amount by which each individual slave's clock requires adjustment.

* This can be a positive or negative value.

* The master takes a fault-tolerant average, that is, a subset of clocks is chosen that do not differ from one another by more than a specified amount, and the average is taken from only these clocks.

* If the master fails, then another computer can be elected to take over and function exactly as its predecessor.

#### 4.2.4 The Network Time Protocol (NTP)

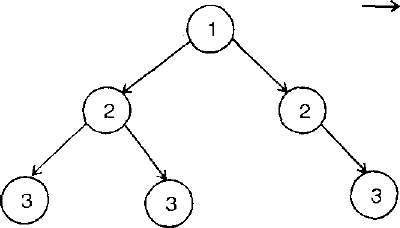
* The NTP defines architecture for a time service and a protocol to distribute time information over the internet.
* The NTP service is provided by a network of servers located across the Internet.

* Primary servers are connected directly to a time source such as a radio .lock receiving UTC; secondary servers are synchronized, ultimately, with primary servers.

* The servers are connected in a logical hierarchy called a synchronization subnet whose levels are called strata.

* Primary servers occupy stratum 1, stratum 2 servers are secondary servers that are synchronized directly with primary servers; stratum 3 servers are synchronized with stratum *2* servers and so on.

* The clocks belongings to servers with high stratum numbers are able to be less accurate than those with low stratum numbers, because errors are introduced at each level of synchronization.

 Denote synthesization control

1, 2, 3 - denote strata

***Figure 3.14: An example Synchronization subnet in ISTP***

NTP servers synchronize with one another in one of three modes:

###  Multicast mode

* It is intended for use on a high-speed LAN.
* One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay.

###  Procedure call mode

* In this mode, one server accepts requests from other computers, which it processes by replying with its timestamp.

* This mode is suitable where higher accuracies are required than multicast mode.

* For example, file servers on the same or a neighboring LAN, which needs to keep accurate timing information for file accesses, could contact a local server in procedure call mode.

###  Symmetric mode

* It is intended for use by the servers that supply time information in LANs and by the higher levels of the synchronization subnet, where the higher accuracies are to be achieved.

* A pair of server operating in symmetric mode exchange messages bearing timing information.

* Timing data are retained as part of an association between the servers that is maintained in order to improve the accuracy *of their* synchronization over time.

* In all modes, messages are delivered unreliably, using the standard UDP Internet transport protocol.

### 4.3 LOGICAL TIME AND LOGICAL CLOCKS

* It is impossible to synchronize clocks perfectly across a distributed system.

* We cannot, in general, use physical time to find out the order of any arbitrary pair of events occurring within it.

* We can use a scheme that is similar to physical causality, but that applies in distributed systems, to order some of the events that occur at different processes.

* This ordering is based on two simple points:

* + If two events occurred at the same process Pi (i = 1, 2, 3 . . . N), then they occurred in the order in which Pi observes them.

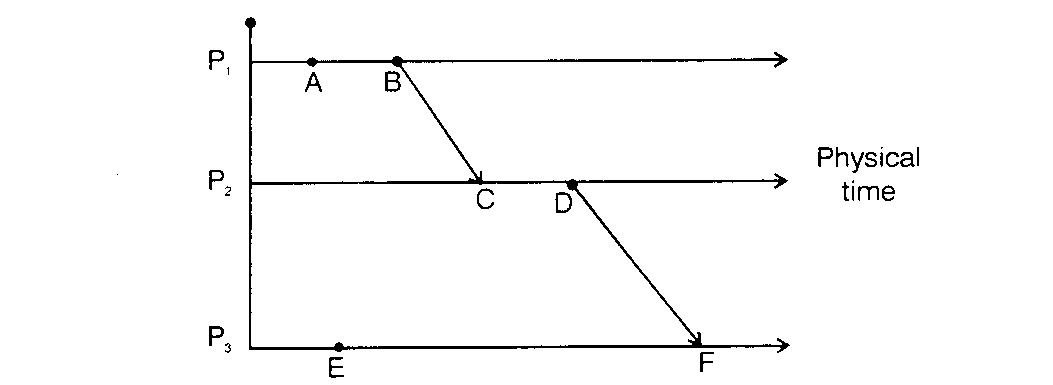
* + Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

* The partial ordering obtained by generalizing these two relationships is called the happened-before relation (or) potential causal ordering.

* The happened-before relation, denoted by is —> defined as follows:

HB1 If з process Pi : e—> i ( e1, then e —> e1

 HB2: : For any message m, send (m) -> receive (m)





H

B

3

:

I

f

e

,

e

1

a

n

d

e

11

a

r

e

e

ven

t

s

s

uc

h

t

ha

t

e

—

>

e

1

a

n

d

e

1

—

>

e

1

1

,

t

h

e

n

e

-

>

e

11

.

T

h

e

r

e

l

at

i

o

n

*—*

*>*

i

s

i

ll

u

s

t

r

a

t

e

d

i

n

f

i

gu

r

e

3

.

1

5

,

f

o

r

t

h

e

ca

s

e

o

f

t

h

r

e

e

p

r

o

ce

s

s

e

s

;

P

,

1

P

2

a

n

d

P

3

.

#### *Figure 3.15: Events occurring at three processes*

* It can be seen that a b, since the events occur in this order at process p| (a —> 1 b) and similarly c —> d.

* Furthermore b —> c, since these events are involved in the sending and receiving message mt, and similarly d —> f. combining these relations, we may also say that a —> f.

* It can also be seen that not a l l events are related by the relation
* For example, a and e are not related with, since they occur at different processes, and there is no chain of messages intervening between them.

* So the events a and e are concurrent and it can be represented by a || e.

### 4.3.1 Logical clocks

* Lamport invented a simple mechanism, by which the happened-before ordering can be captured numerically, called a logical clock.

* A lamport logical clock is a monotonically increasing software counter.

* Each process Pi keeps its own logical clock, Li which it uses to appl y time stamps to event s.

* The timestamp of event e at Pi. is denoted by Li (e).

* Processes update their logical clocks and transmit the values of their logical clocks in message as follows:
  + LC1: L. is incremented before each event is issued at process P*i* : Li – Li+ l

* + LC2: a) when a process P. sends a message m, it piggybacks an m the value t = Li.

b) On receiving (m, t), a process Pj computes Lj = max (Lj , t) and then applies LC1, before time-stamping the event receive (m).

* If e—>e1 then L(e) < L(e1 ). But the converse is not true. That is. If L(e) < L(e1), then we cannot infer that e e1.

\*

B

Ph

y

s

c

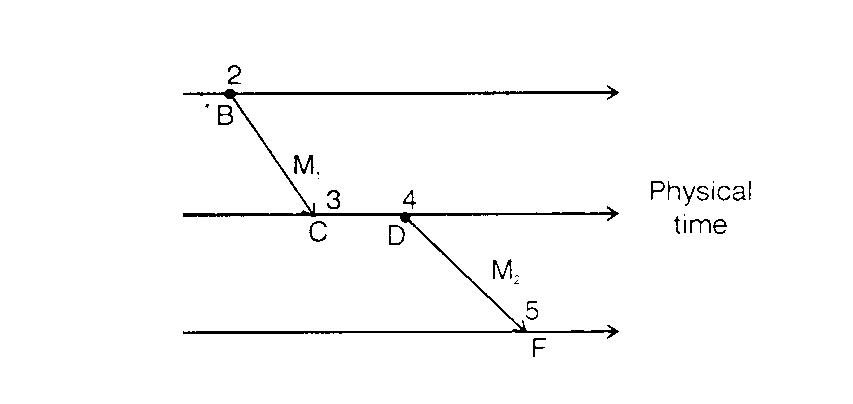
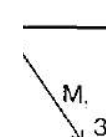
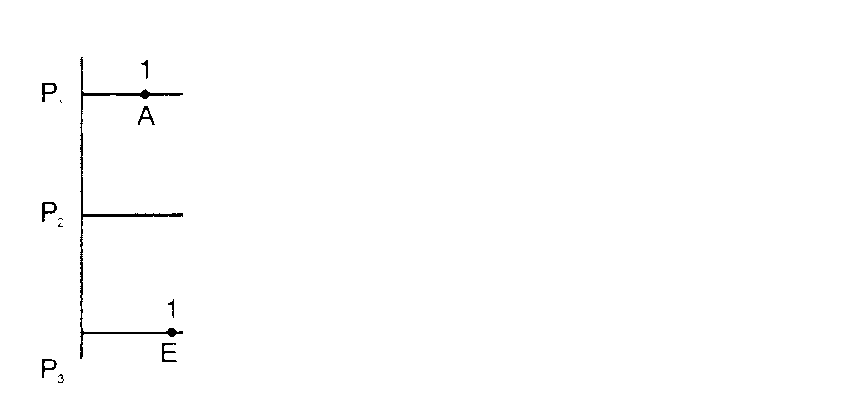
a

m

e

M

C



i

l

t

i

#### *Figure 3.16: Lamport rimestamps for the events*

 Each processes P 1 , P2&P3 has its logical clock i n i t i a l i z e d to 0 the clock values given are those immediately after the event to which they are adjacent. E.g. L (b ) > L(e) but b || e.

####  Totally ordered logical clocks

* Some pairs of distinct events, generated by different processes, have numerically identical lamport timestamps.

i

* However, we can create a total order on events by taking int o account the identifiers of the processes at which events occur.

* If e is an event occurring at Pi with local timestamp Ti, and e1 is an event occurring at Pi with local timestamp T .
* We define the global logical timestamps for these events to be (T I, i) and (T j, j) respectively.

# 4.3.2 Vector clocks

* Vector clocks were developed to overcome the shortcoming of lamport’s clocks, the fact that from L(e) < L(e1) we cannot conclude that e->e1.

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

* Each process keeps its own vector clock ViP which it uses to timestamp local events.

* The rules for updating the clocks are as follows:

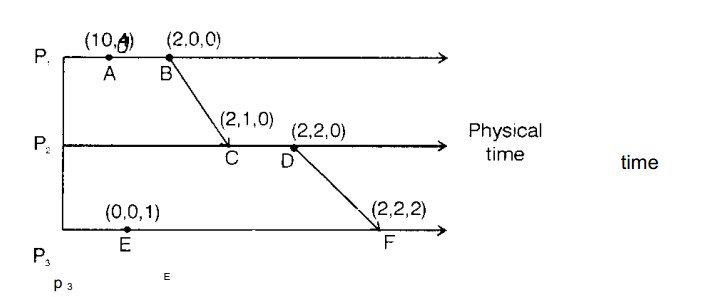
o VC1: Initially, Vi(j) = 0, for i , j = 1 , 2 , . . . N.

o VC2: Just before *pt* timestamps an event, it sets Vi[i] = Vj[ i ] + 1

o VC3: P. includes the value t = Vi in every message it sends

o VC4: when P. receives a timestamp t in a message, it sets Vi[j] = max (Vi [ j] , t [ j ] ) , for j = 1, 2 . . . N. Taking the component wise maximum of two vector timestamps in this way is known as a merge operation.

For a vector clock Vi, Vi[i], is the number of events that P. has time stamped, and vi[j] such that (j≠i) is the number of events that have occurred at Pj that P] has potentially been affected by.



## *Figure 3.1 7: Vector timestamps for the events*

***IT T82 Distributed Computing***

* If e e1 then V (e) < V (e1). Also the converse is true. That is, if V(e) < V(e1) then

**Disadvantage**

* Amount of storage and message payload that is proportional to N.

## 4.4 Global States

* On many occasions, it is useful to know the global state in which a distributed system is c residing.

* The global state of a distributed system consists of the local state of each process, to the messages that are currently in transit, i. e. that have been sent but not delivered.

* Knowing state of a distributed system may be useful in detecting the following.
  + Distributed deadlock detection

* + Distributed garbage object detection

* + Distributed termination detection

* + Distributed debugging.

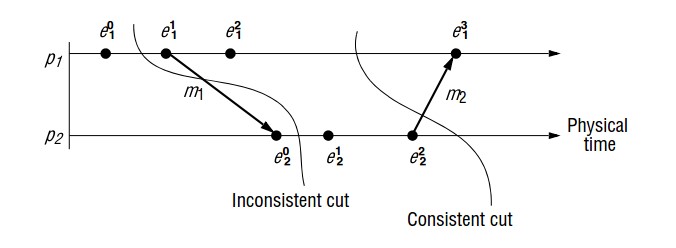
* The global history is defined as the union of the individual process histories H=h0 Ụ h1 Ụ ….. Ụ hn-1

* Mathematically any set of states of the individual processes can be taken to form a global state (s1 s2 . . . sN).
* The notion of global state can be graphically represented by what is called as a cut which represents the last event that has been recorded for each process.
* Consider the events occurring at processes P1 and P2 as shown in figure 3.18, it has two cuts.

* The leftmost cut is inconsistent. This is because, at P2 it includes the receipt of the message m1 but at P1, it does not include the sending of that message.
* This is showing an ‗effect‘without a ‗cause‘.

* By contrast, the rightmost cut is consistent. It includes both the sending and the receipt of message m1 and the sending but not the receipt of the message m 2.
* That is consistent with the actual execution because, the message may take some time to arrive.

Dept. of Information Technology, SMVEC Page 33



### *Figure 3.18: Cuts*

* 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 eєC, f e => = fєC.

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

* A run is a total ordering of all the events in a global history that is consistent with each local history's ordering, i(i=l, 2 . . . N).

* A linearization a consistent run is an ordering of the events in a global history with this happened before relation on H.

* A state S1 is reachable from a state S if there is a linearization that passes through S and then S1.

##  Snapshot algorithm of candy and lamport

* This algorithm is used for determining global states of a distributed system. The goal of the algorithm is to record a set of process and channel states for a set of processes Pi (i = l, 2. . . N ).

* The algorithm assumes that:

* + Neither channels nor processes fail; communication is reliable so that every message sent is eventually received, exactly once.

* + Channels are unidirectional

* + The graph of processes and channels is strongly connected.

* + Any process may initiate a global snapshot at any time.

* + The processes may continue their execution and sent and receive normal messages while the snap shot takes place.
  + For each process Pi, the incoming and outgoing channels are used for receiving and sending messages respectively.

* + The algorithm proceeds through the use of special marker messages.

* + The marker has a dual role: as a prompt for the receiver to save its own state, if it has not already done so; and as a means of determining which messages to include in the channel state.

* + The algorithm is defined through two rules; the marker receiving rule and the marker sending rule.

* **The *marker sending rule***obligates processes to send a marker after they have recorded their state, but before they send any other messages.

* The ***marker receiving rule***delegates a process that has not recorded its state to do so. In that case, this is the first marker that it has recorded.

* It notes which messages subsequently arrive on the other incoming channels.

* When a process that has already saved i t s state receives a marker on another channel, it records the state of that channel as the set of messages it received on it since it saved on its state.

 **Snapshot Algorithms:**

* 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 the 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)

* A process is said to have finished i t s parts of the algorithm when it has received a marker along each of its incoming channels and processed each one.

* At that point, it’s recorded local state, as well as the state it recorded for each incoming channel, can be collected and sent to the process that initiated the snapshot.

## 4.5 DISTRIBUTED DEBUGGING

* One example which we have already discussed is that each set of processes Pi, has a variable xi.

* The safety condition required is |x i -xj| < ∆ (i, j = 1, 2 . . . , N); this constraint is to be met even though a process may change the value of i t s variable at any time.

* Another example in distributed system controlling is a system of pipes in a factory where we are interested in whether all the valves were open at the sometime.

* In these examples, we cannot generally observe the values of the variables or the states of the valves simultaneously.

* The challenge is to monitor the system's execution overtime - to capture trace information rather than a single snapshot - so that we can establish post hoc whether the required safely condition was or may have occurred.

* Chandy and Lamport's snapshot algorithm collects state in a distributed fashion, and the processes in the system send the state they gather to a monitor process for collection.

* The monitor assembles globally consistent states from what it receives. It is considered that the monitor is outside the system, observing its execution.

* Our aim is to determine cases where a given global state predicate φ was *definitely True* at some point in the execution we observed, and eases where it *was possibly True.*

* The notion possibly arises as a natural concept because we may extract a consistent global state S from an executing system and find that φ(S) is *True.*

* No single observation of a consistent global state allows us to conclude whether a non-stable predicate ever evaluated to *True* in the actual execution.

* Nevertheless, we may be interested to know whether they might have occurred, as far as we can tell by observing the execution.

* The notion definitely does apply to the actual execution and not to a run that we have extrapolated from it.

* It is possible to evaluate whether 0 was definitely *True* by considering all linearizations of the observed events.

* The definition *of possibly φ* and *definitely* (J), for a predicate *φ* in terms of linearizations of H, the history of the system's execution is given below.

* + **possibly** *φ* The statement possibly *φ* means that there is a consistent global state S through which a linearizations of H passes such that *φ* (S) is True.

* + **definitely** *φ*: The statement definitely φ means that for all linearizations L of H, there is a consistent global state S through which L passes such that *φ* (S) is True.

###  Collecting the state

* The observed processes Pi ( i = l , 2 . . . , N) send their initial state to the monitor process i n i t i a l l y and thereafter from time to time, in state messages.

* The monitor process records the state messages from processes Pi in a separate queue Q. for each i= 1, 2

.... N.

* The activity of preparing and sending state messages may delay the normal execution of the observed processes, but it does not otherwise interfere with i t .

* There is no need to send the state except initially and when it changes.

* There are two ways to reduce the state-message traffic to the monitor.

o First, the global state predicate may depend only on certain parts of the processes‘ states.

* For example, it may depend only on the states of the particular variables.

* So the observed processes need only send the relevant state to the monitor process.

* Second, they need only send their state at times when the predicate *φ* may become true or cease to be true.
* There is no point in sending changes to t h e state that does not affect the predicate's value.

#### 4.5.1 Observing consistent global state

* The monitor must assemble consistent global states against which it evaluates <J>. Let us consider two processes P. and P, with variables: x1 and x2, respectively.

* Initially x1= x2 = 0, the requirement is |x1 – x2| < 50.

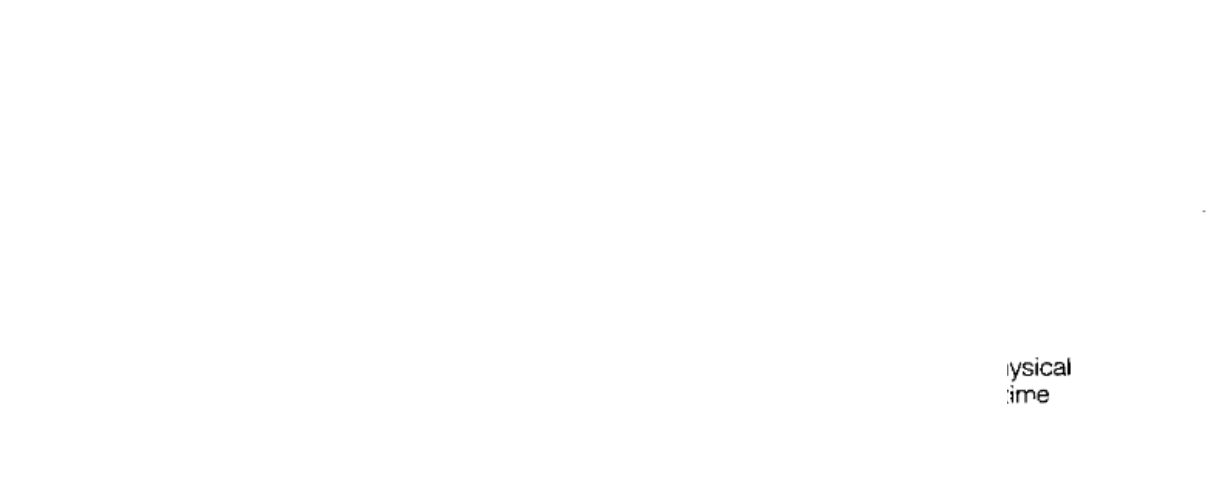
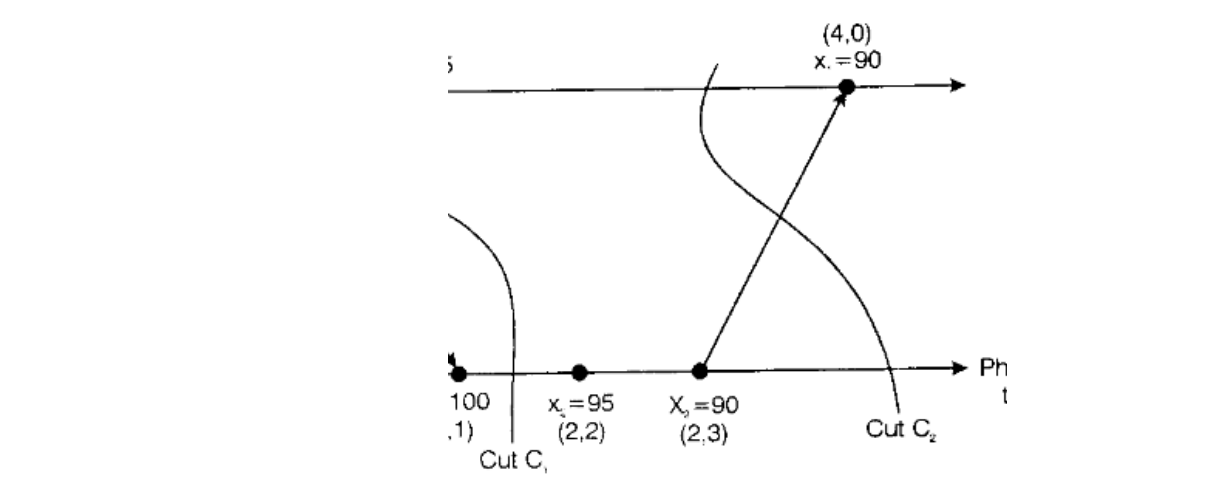
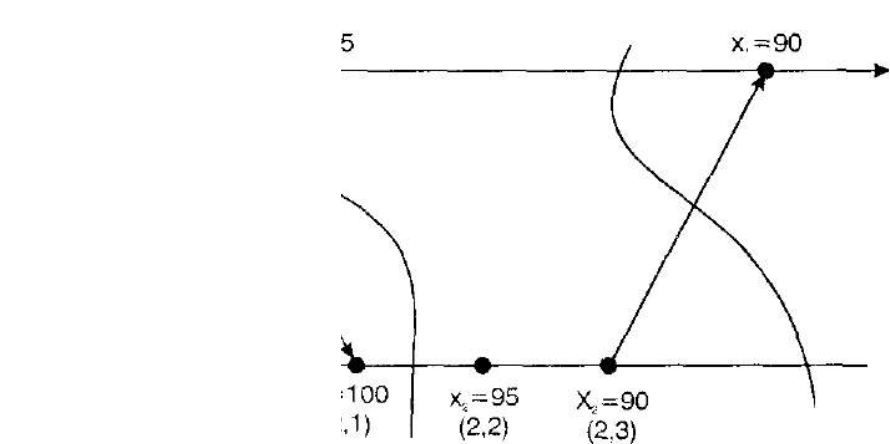
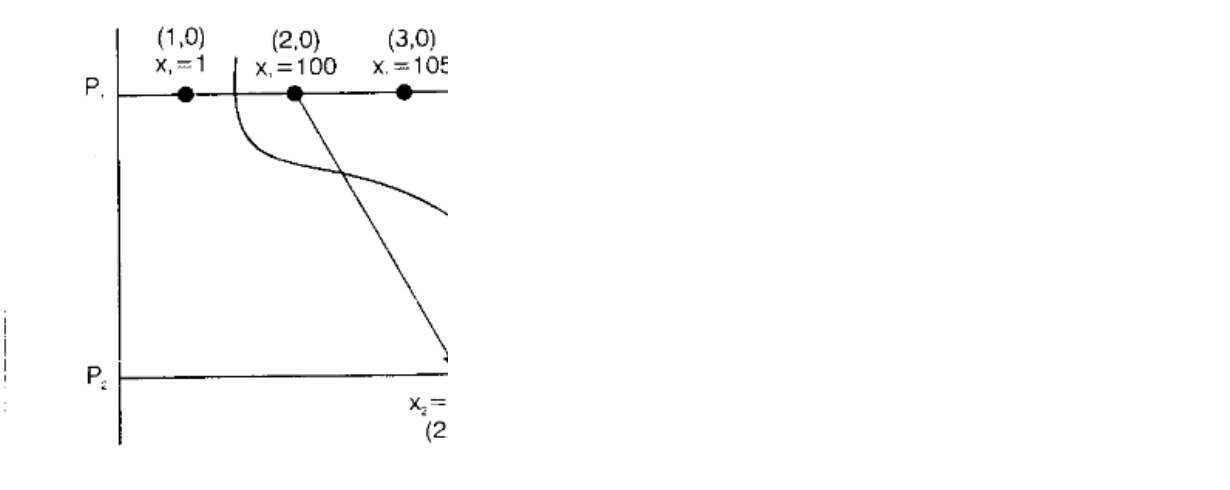
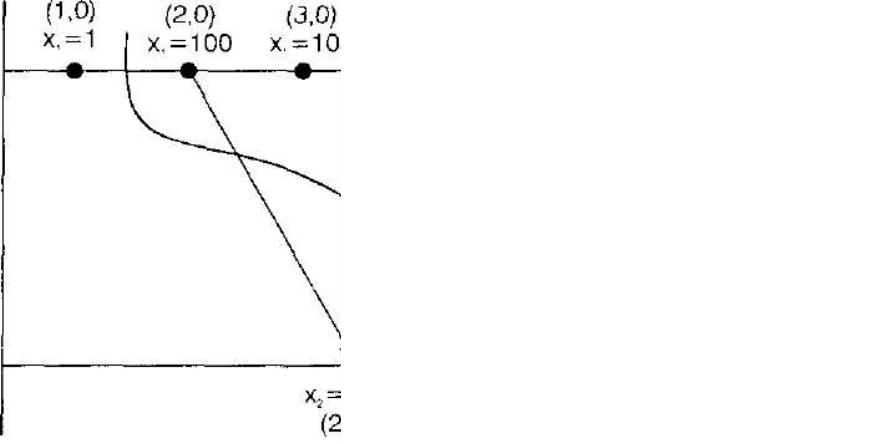
* The processes make adjustments to their variables, but large adjustments cause a message containing the new value to be sent to the other process.

* When either of the processes receives an adjustment message from the other, it sets its variable equal to the value contained in the message.

* Whenever one of the processes P1 or P2 adjusts the value of its variable (whether it is small adjustment or large one), it sends the value in a state message to the monitoring process.

* The latter keeps the state messages in the per-process queues for analysis. If the monitor processes used values from inconsistent cut C1, in figure 3.19, then it would find that x = 1, x, = 100, breaking the constraint |x - x,| <50.

* This state of affairs never occurred, instead, values from the consistent cut C2 show x1 = 105 and x2= 90.



##### Figure 3.19: lector timestamps and variable values for the execution of figure 3.18

* In order that the monitor can distinguish consistent global states from inconsistent global states, the observed processes enclose their vector clock values with their state messages.

* Each queue Qi is kept ordered in sending order, which can immediately be established by examining ilh component of the vector timestamps.
* Of course, the monitor process may deduce nothing about the ordering of states sent by different processes from their arrival order, because of variable message latencies.

* It must instead examine the vector timestamps of the state messages.

* Algorithms to evaluate possibly φ and definitely φ are given below; Evaluate possibly <|> for global

history H of N processes L= 0;

States ={(s10,s20,F,sn0)};

While ( *φ*( S) = False for all S є States

L=L + 1;

Reachable = {S1: S1 reachable in H from some S €States A level (S1) = L};

States = Reachable;

end while

output *"possibly φ”*

* Evaluate definitely *φ* for global history H of N processes

L=0;

If φ (*(*s10,s20,F,sn0*.))* then States = {} else

St ates (s10,s20,F,sn0 )}; while (States ≠{})

L=L+1;

Reachable = {S1: S1 reachable in H from some S States level (S1) = L}; States = {S Reachable: φ(S) = False} end while

output *"definitely φ* "

#### 4.5.2 Evaluating possibly *φ* and definitely *φ* in synchronous systems

* The algorithms discussed in previous section will work in an asynchronous system: i have made no timing assumptions.

* But the price paid for this is that the monitor may examine a consistent global state S = (s1, s2, F,sn) for which any two local states si and sj occurred arbitrarily long time apart in the actual execution of the system.

* Our requirement, by control is to consider onl y those global states that the actual execution coul d in principal *Y* traversed.

* In a synchronous system, suppose that the processes keep their physical clocks inter synchronized within a known bound, and that the observed processes provide by timestamps as well as vector timestamps in their messages.

* Then the monitor process considers only those consistent global states whose local states could possibly have simultaneously, given the approximate synchronization of clocks.

* With good enough synchronization, these will number many less than all globally consistent states.

* Let us now discuss an algorithm to exploit synchronized clock in this way. We ; that each observed process P; ( i = l , 2 , . . . N ) and the monitor process, which we shall keeps a physical clock Ci ( i = 0, l , . . . , N ).

* These are synchronized to within a known bound that is, at the same real time:

| C i (t) - C j (t) i < D f o r i, j - l, 2. . . N

* The observed processes send both their vector time and physical time with their state messages to the monitor process.

* The monitor process now applies a condition that not only tests for consistency of a global state S = ( s10,s20,F,sn0), but also tests whether each pair of states could have happened at the same real time, given the physical clock values.

* In other words, for i,j= 1,2, . . . , N:

Vsi[i] > V (Sj)[i] and s and s could have occurred at the same real time.

* The first clause is the collection that we used earlier.

* For the second clause, note that P is in the state s. from the time it first notifies the monitor process. Cj(s.), to some later local time Li(si), say, when the next state transition occurs at Pi.

* For si and sj to have obtained at the same real time we thus have, allowing for the bound on clock synchronization:

Ci - or vice versa (swapping i and j) Ci (si) - D Cj(sj) L(Sj) + D - or vice versa(swapping i and j)

* The monitor process must calculate a value for Li(Sj), which is measured against Pi‘s clock.

* If the monitor process has received a state message for pi's next state , then L.(s.) is Ci(si1).

* Otherwise, the monitor process estimates Li(si) as C0 - *max* +- D, where C0 is the monitor's current clock value, and *max* is the maximum transmission time for a state message.

### 4.6 Coordination & Agreement

### DISTRIBUTED MUTUAL EXCLUSION

* Distributed processes often need to coordinate their activities.

* If a collection of processes share a resource or collection of resources, then mutual exclusion is required to prevent interference and ensure consistency when accessing the resources.

* This is the critical section problem, familiar in the domain of operating systems.

**Algorithms for mutual exclusion:**

Our essential requirements for mutual exclusion are as follows.

ME 1: (safety) - At most one process may execute in the Critical Section (CS) at a time

ME 2: (liveness) - Requests to enter and exit the CS eventually succeed.

ME 3: (ordering) - If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

* A *deadlock* would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the CS.

* A *starvation* is the indefinite postponements of entry for a process that has requested i t .

* Condition ME2 implies freedom from both deadlock and starvation. ME3 specifies that the first process be granted access before the second.

**4.6.1 The central server algorithm**:

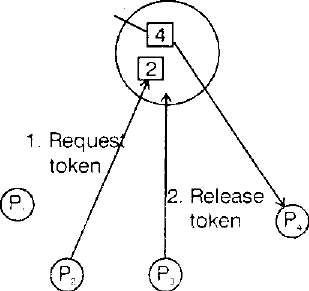
* The simplest way to achieve mutual exclusion is to empty a server that grants permission to enter the critical section.

* To enter a critical section, a process sends a request message to the server and awaits a reply from it .

* The reply constitutes a token signifying permission to enter the critical section.

* If no other process has the token at the time of request, then the server replies immediately, granting the token.

* If the token is currently held by another process, then the server does not reply but queens the request on exiting the critical section, a message is sent to the server, giving it back the token.
* In the figure 3.20, P2's request has been appended to the queue, which already contained P4's request. P3 exit the critical section, and the server removed P4‘s entry and grants permission to enter to critical section.



Q

u

e

u

e

o

f

re

q

u

e

st

s

***Figure 3 .20: Server managing a mutual exclusion token***

### Disadvantage

The server may become a performance bottleneck for the system as a whole.

#### 4.6.2 A Ring based algorithm's

* One of the simplest ways u> arrange mutual exclusion between the M processes without requiring an additional process is to arrange them in a logical ring exclusion is conferred by obtaining a token in me form *of d* message passed from process to process in a single direction- clockwise around the ring.

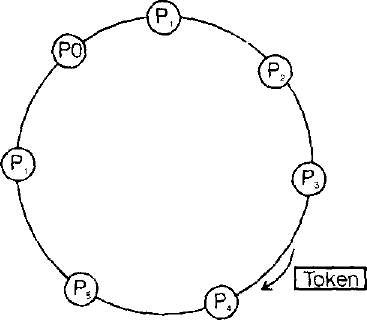
* If a process does not require entering the critical section when it receives the token, then it immediately forwards the token to its neighbour.

* A process that requires the token waits until it receives it but retains it.

* To exit the critical section, the process sends the token on to its neighbour. This algorithm continuously consumes network bandwidth

* The delay experienced b y a process requesting entr y to CS is between 0 messages and N messages.

***IT T82 Distributed Computing***



##### Figure 3.21: A ring of processes transferring a mutual exclusion token

4.6.3 **An algorithm using multicast and logical clocks:**

* The basic idea is that processes that require entry to a critical section multicast a request message and can enter it only when all the other process have replied to this message.

* The processes P1, P2 ... PN bear distinct numeric identifiers.

* They are assumed to possess communication channels to one another, and each process Pi keeps a lamport clock, updated according to the rules LC1 and LC2.

* Messages requesting entry are of the form < T, Pi >, where T is the sender's timestamp and Pi is the sender's identifier.

* Each process records its state of being outside the CS (RELEASED), waiting entry (WANTED) or being in the CS (HELD) in a variable state. The protocol is given below.

*On* initialization

state = RELEASED To enter the section

state = WANTED

Multicast request to call processes;

T:= request's timestamp;

wait until (number of replies received = (N-l) ;

state= HELD

on receipt of a request < Ti ,Pi > at Pj (i≠j)

if (state = HELD or (state = WANTED and (Ti, Pj) < (Ti, Pj) then

queue request from Pi without replying; else

reply immediately to Pj

endif

Dept. of Information Technology, SMVEC Page 43

To exit the critical section state: = RELEASED reply to any queued requests;

* If a process requests entry and the state of all other processes is RELEASED, then processes will reply immediately the request and the requestor will obtain entry.

* If some process is in state HELD, then that process will not reply to requests until finished with the CS and so the requester cannot gain entry in the meantime.

* If two or more process request entry at the same time then whichever process's request bears the lowest timestamp will be first to collect N-l replies, granting it entry next.

* If the request equal timestamp the requests are ordered according to the processes corresponding identifiers.
  + More expensive algorithm, in terms of bandwidth consumption.

* + Improved performance.

* To illustrate the algorithm, consider a situation involving 3 processes, P1 ,P2 and P3 shown in figure 3.22.
* Let us assume P3 is not interested in entering the critical section and that P1: and P2 request entry concurrently. The timestamp of P 1’s request is 4 1and that of P2‘s 34.

* When P3 receives their requests, it replies immediately.

* When P2 receives P1‘s require finds its own request has the lower timestamp, and so does not reply, holding P1 off.

* However P1 finds P2‘s request has a lower timestamp than that its own request and immediately. On receiving this second reply, P2 can enter the critical section.

* When P2 the critical section, it will reply to P1‘s request and so grant it entry.

* Gaining entry takes 2(N-1) messages in this algorithm: (N-1) to multicast the request, followed by N-1 replies. *Figure 3.22: Multicast Synchronization*

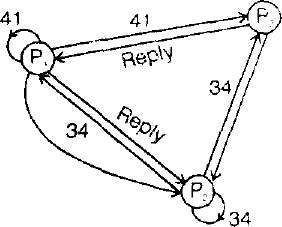
R

e

p

4

1



#### 4.6.4 Maekawa's voting algorithm

* **Maekawa** observed that in order for a process to enter a critical section, it is not needed for all of its pears to grant it access.

* Processes need only obtain permission to enter subsets of their peers, as long as the subsets used by any two processes overlap.

* A Candidate process must collect sufficient votes to enter.

* Processes in the intersection of two sets of voters ensure the safety property ME 1, that at most one process can enter the critical section, by casting their votes for only one candidate.

* Maekawa associated a voting set Vi with each process Pi (i = 1 , 2 . . . N ) , where Vi < (P1 , P2...PN).

* The sets V are chosen so that for all i, j = 1, 2 . . . . N : P€Vi

* V∩V≠φ: there is at least one common member of any two voting sets.

* |Vi| = K: to be fair, each process has a voting set of the same size. Each process P. is contained in M of the voting sets V.

**Maekawa's algorithm:**

On initialization

state: = RELEASED; voted : =FALSE;

for Pi to enter the critical section state - W ANTED;

multicast request to all process in V - {Pj };

wait until (number of replies received = (K-L));

state = HELD;

On receipt of request from Pi at Pj (i ≠ j) if (state = HELD or voted = TRUE) then Queue request from P:

without replying;

else

send reply to P; voted : = TRUE;

end if

for P to exit the critical section

i

state = RELEASED;

Multicast release to all processes in Vj- {P,}; On receipt of a release from Pj at Pi(i != j)

if (queue of requests is non-empty) then

Remove head of queue (say Pk); Send reply to

Pk;

voted = TRUE;

else voted = FALSE;

end if

* To obtain entry to the critical section, a process Pj sends request messages to all other K-l members of Vi, P cannot enter the critical section until it has received all K-l request messages.

* When a process Pj in Vi receives Pi‘s request message, it sends a reply message immediately, unless either i t s state is HELD or it has already replied (voted) since it received a release message.

* Otherwise, it queues the request message by does not yet when a process receives a release message, it removes the head of its queue of outstanding requests and sends a reply message in response to it.

* To leave the critical section, P1‘s release messages to all the other K-l members of Vi.

## Drawback

* The algorithm is deadlock pure. Consider 3 process Pp p, and p3 with V1 = {P1,P2, P3} and V3=(P3,,P1).

* If the three processes concurrently request entry to the critical section, then it is possible for P] to reply to P; but hold off Pj for P2 to reply to P3 but ho P 1 and for P3 to reply to P1 but hold off P2.

**4.7 Election: choosing a unique process for a particular role**

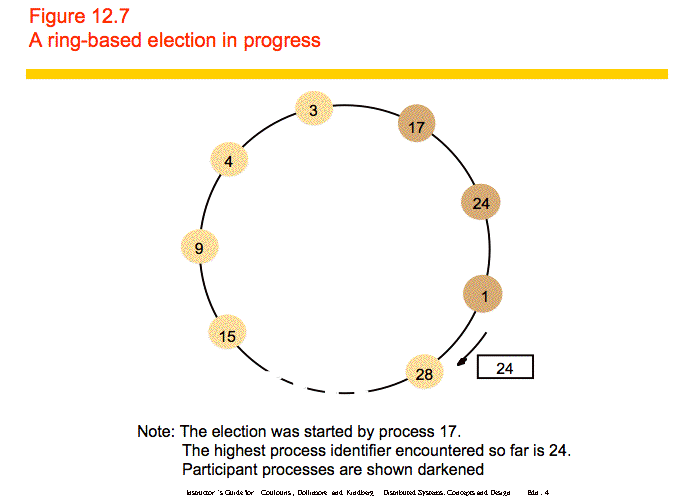
An election algorithm determines which process will play the role of coordinator or server. All processes need to agree on the selected process. Any process can start an election, for example if it notices that the previous coordinator has failed. The requirements of an election algorithm are as follows:

* Safety: Only one process is chosen -- the one with the largest identifying value. The value could be load, uptime, a random number, etc.
* Liveness: All process eventually choose a winner or crash.

**Ring-based**

Processes are arranged in a logical ring. A process starts an election by placing its ID and value in a message and sending the message to its neighbor. When a message is received, a process does the following:

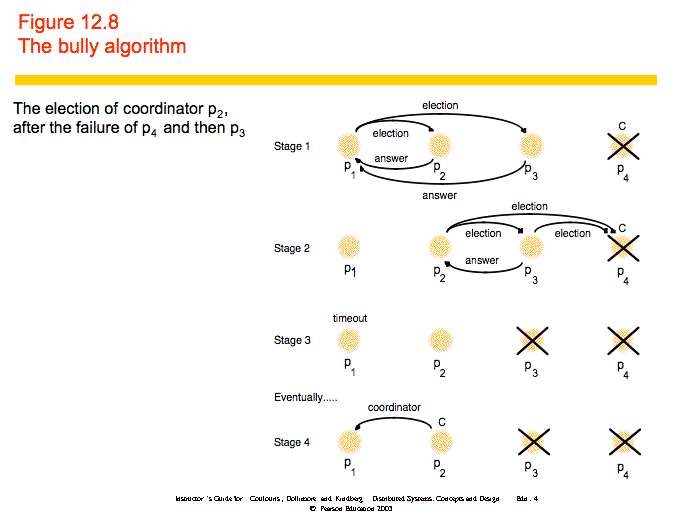
* If the value is greater that its own, it saves the ID and forwards the value to its neighbor.
* Else if its own value is greater and the it has not yet participated in the election, it replaces the ID with its own, the value with its own, and forwards the message.
* Else if it has already participated it discards the message.
* If a process receives its own ID and value, it knows it has been elected. It then sends an elected message to its neighbor.
* When an elected message is received, it is forwarded to the next neighbor.



Safety is guaranteed - only one value can be largest and make it all the way through the ring. Liveness is guaranteed if there are no failures. However, the algorithm does not work if there are failures.

**Bully**

The bully algorithm can deal with crash failures, but not communication failures. When a process notices that the coordinator has failed, it sends an election message to all higher-numbered processes. If no one replies, it declares itself the coordinator and sends a new coordinator message to all processes. If someone replies, it does nothing else. When a process receives an election message from a lower-numbered process it returns a reply and starts an election. This algorithm guarantees safety and liveness and can deal with crash failures.



## Consensus

All of the previous algorithms are examples of the consensus problem: how can we get all processes to agree on a state? Here, we look at when the consensus problem is solvable.

The system model considers a collection of processes pi (i = 1, 2, ..., N). Communication is reliable, but processes may fail. Failures may be crash failures or byzantine failures.

The goals of consensus are as follows:

* Termination: Every correct process eventually decides on a value.
* Agreement: All processes agree on a value.
* Integrity: If all correct processes propose the same value, that value is the one selected.

We consider the Byzantine Generals problem. A set of generals must agree on whether to attack or retreat. Commanders can be treacherous (faulty). This is similar to consensus, but differs in that a single process proposes a value that the others must agree on. The requirements are:

* Termination: All correct processes eventually decide on a value.
* Agreement: All correct processes agree on a value.
* Integrity: If the commander is correct, all correct processes agree on what the commander proposed.

If communication is unreliable, consensus is impossible. Remember the blue army discussion from the second lecture period. With reliable communication, we can solve consensus in a synchronous system with crash failures.

We can solve Byzantine Generals in a synchronous system as long as less than 1/3 of the processes fail. The commander sends the command to all of the generals and each general sends the command to all other generals. If each correct process chooses the majority of all commands, the requirements are met. Note that the requirements do not specify that the processes must detect that the commander is fault.

It is impossible to guarantee consensus in an asynchronous system, even in the presence of 1 crash failure. That means that we can design systems that reach consensus most of the time, but cannot guarantee that they will reach consensus every time. Techniques for reaching consensus in an asynchronous system include the following:

* Masking faults - Hide failures by using persistent storage to store state and restarting processes when they crash.
* Failure detectors - Treat an unresponsive process (that may still be alive) as failed.
* Randomization - Use randomized behavior to confuse byzantine processes.