## Unit – IV

Introduction - Clocks, events and process states - Synchronizing physical clocks- Logical time and logical clocks - Global states – Coordination and Agreement - Introduction - Distributed mutual exclusion – Elections - Transactions and Concurrency Control– Transactions -Nested transactions – Locks – Optimistic concurrency control - Timestamp ordering – Atomic Commit protocols -Distributed deadlocks – Replication – Case study – Coda.

## Introduction

Time is an important and interesting issue in distributed systems, for several reasons. First, a computer's clock should be synchronized with an external, source to measure time accurately, with respect to occurrence of events.

Second, algorithms for clock synchronization to maintain the consistency of distributed data, checking the authenticity of a request sent to a server and eliminating the processing of duplicate updates have been developed.

Measuring time can be problematic due to the existence of multiple frames of reference. For example, an observer on the Earth and an observer travelling away from the Earth in a spaceship will disagree on the time interval between events, the more so as their relative speed increases.

The notion of physical time is also problematic in a distributed system. For example, in object oriented systems, establishing whether references to a particular object no longer exist – whether the object has become garbage, requires observations of the states of processes and of the communication channels between processes.

## Clocks, events and process states

A distributed system to consist of a collection **ρ** of N processes $\mathbf{p_i}$ , i = 1, 2, ..N. Each process executes on a single processor, and the processors do not share memory.

Each process $\mathbf{p_i}$ in **ρ** has a state $\mathbf{s_i}$ that, in general, it transforms as it executes. The process's state includes the values of all the variables within it. Its state may also include the values of any objects in its local operating system environment that it affects, such as files.

We assume that processes cannot communicate with one another in any way except by sending messages through the network.

As each process $p_i$ executes it takes a series of actions, each of which is either a message **send** or **receive** operation, or an operation that transforms $\mathbf{p_i}$'s state – one that changes one or more of the values in $\mathbf{s_i}$ .

An **event** is defined as the occurrence of a single action that a process carries out as it executes. The sequence of events within a single process $\mathbf{p_i}$ can be placed in a single, total ordering, which we denote by the relation $\rightarrow_i$ between the events. That is, e $\rightarrow$ i e' if and only if the event e occurs before e' at $\mathbf{p_i}$ .

The history of process $\mathbf{p_i}$ can be defined as the series of events that take place within it, ordered as we have described by the relation $\rightarrow$ i:

$$history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

## Clocks

Computers each contain their own physical clocks. These clocks are electronic devices that count oscillations occurring in a crystal at a definite frequency, and typically divide this count and store the result in a counter register.

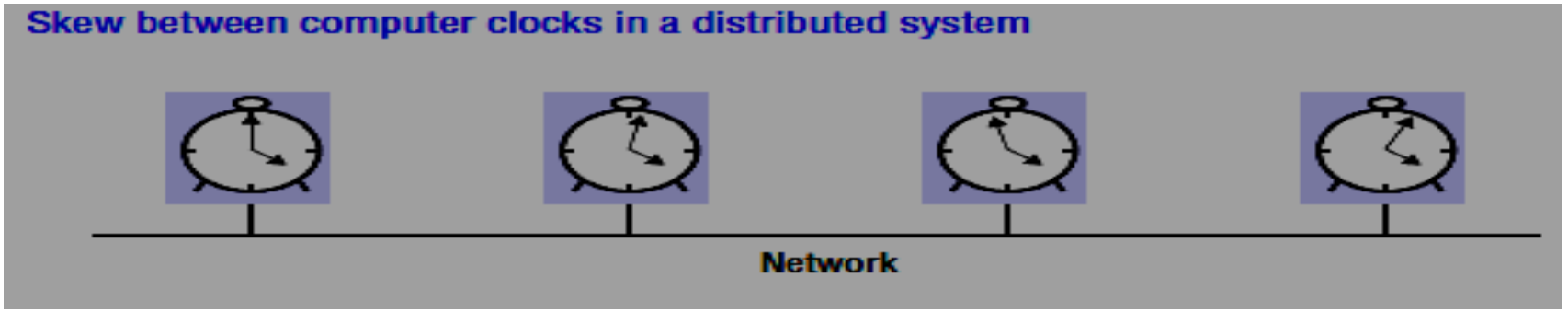
Clock devices can be programmed to generate interrupts at regular intervals in order that, for example, time slicing can be implemented.

The operating system reads the node's hardware clock value, $H_i(t)$, scales it and adds an offset so as to produce a software clock $C_i(t) = \alpha H_i(t) + \beta$ that approximately measures real, physical time t for process $\mathbf{p_i}$ .

In general, the clock is not completely accurate, so $C_i(t)$ will differ from t. Nonetheless, if $C_i$ behaves sufficiently well, its value can be used to timestamp any event at $\mathbf{p_i}$ .

Successive events will correspond to different timestamps only if the **clock resolution** – the period between updates of the clock value – is smaller than the time interval between successive events.

# Clock skew and clock drift




Computer clocks, like any others, tend not to be in perfect agreement. The instantaneous difference between the readings of any two clocks is called their **skew**.

Also, the crystal-based clocks used in computers are, like any other clocks, subject to clock drift, which means that they count time at different rates, and so diverge.

\ A clock's **drift rate** is the change in the offset (difference in reading) between the clock and a nominal perfect reference clock per unit of time measured by the reference clock.

## Coordinated Universal Time

Computer clocks can be synchronized to external sources of highly accurate time. The most accurate physical clocks use atomic oscillators, whose drift rate is about one part in $10^{13}$. The output of these atomic clocks is used as the standard for elapsed real time, known as **International Atomic Time.**

**Coordinated Universal Time** – abbreviated as **UTC**– is an international standard for timekeeping. It is based on atomic time, but a so-called 'leap second' is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time.

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

## Synchronizing physical clocks

In order to know at what time of day events occur at the processes in our distributed system **ρ** – for example, it is necessary to synchronize the processes' clocks, $C_i$ , with an authoritative, external source of time. This is **external synchronization**.

If the clocks *Ci* 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 necessarily synchronized to an external source of time. This is **internal synchronization.**

These two modes of synchronization are more closely defined 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. Another way of saying this is that the clocks $C_i$ are accurate to within the bound D.

- **Internal synchronization:** For a synchronization bound D > 0 , $|C_i(t) - C_j(t)| < D$ for i, j = 1, 2 …N, and for all real times t in I. Another way of saying this is that the clocks Ci agree within the bound D

**Monotonicity** is the condition that a clock C only ever advances:

$$t' > t \Rightarrow C(t') > C(t)$$

Monotonicity can be achieved despite the fact that a clock is found to be running fast. We need only change the rate at which updates are made to the time as given to applications. This can be achieved in software without changing the rate at which the underlying hardware clock ticks.

A hybrid correctness condition requires that a clock obeys the monotonicity condition, and that its drift rate is bounded between synchronization points, but the clock value can be allowed to jump ahead at synchronization points.

A clock that does not keep to whatever correctness conditions apply is defined to be **faulty**. A clock's **crash failure** is said to occur when the clock stops ticking altogether; any other clock failure is an **arbitrary failure**.

A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broke the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000;

Clocks **do not have to be accurate to be correct.** Since the goal may be internal rather than external synchronization, the criteria for correctness are only concerned with the proper functioning of the clock's 'mechanism', not its absolute setting.

## 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 required to execute each step of a process.

One process sends the **time t** on its local clock to the other in a **message m**. In principle, the receiving process could set its clock to the **time t + Ttrans** , where **Ttrans** is the time taken to transmit m between them. Unfortunately, **Ttrans** is subject to variation and is unknown.

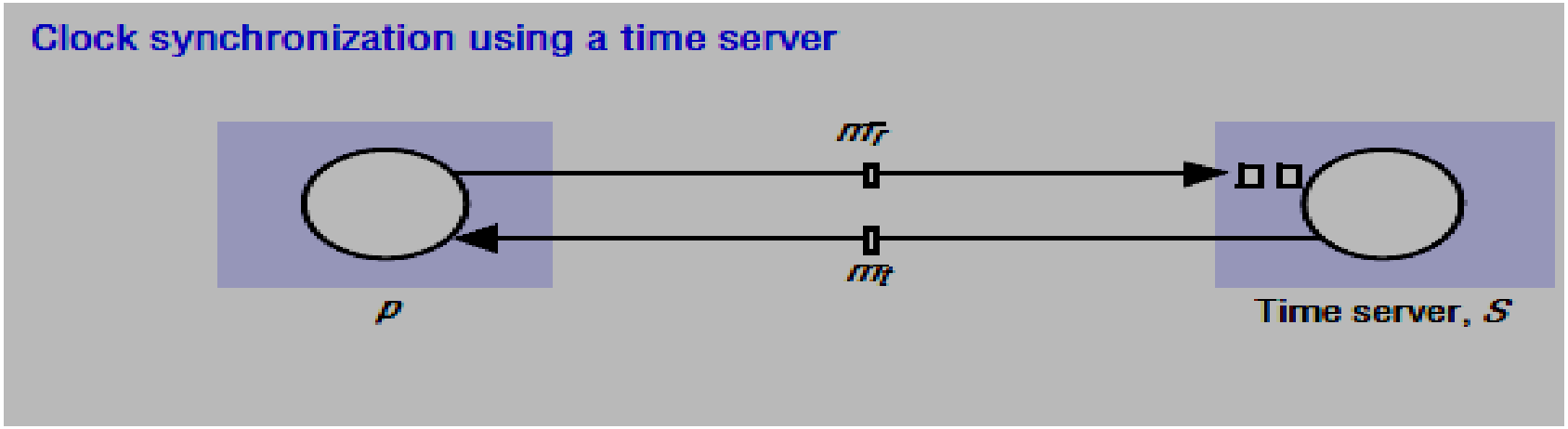
Nonetheless, there is always a minimum transmission time, **min**, that would be obtained if no other processes executed and no other network traffic existed; **min** can be measured or conservatively estimated.

Most distributed systems found in practice are asynchronous: the factors leading to message delays are not bounded in their effect, and there is no upper bound **max** on message transmission delays.

For an asynchronous system, we may say only that **Ttrans = min + x** ,where **x ≥ 0**. The value of x is not known in a particular case, although a distribution of values may be measurable for a particular installation.

## Cristian's method for synchronizing clocks

Cristian suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally. Upon request, the server process S supplies the time according to its clock, as shown in Figure.



In this method even though there is no upper bound on message transmission delays in an asynchronous system, the round-trip times for messages exchanged between pairs of processes are often reasonably short – a small fraction of a second.

He described the algorithm as probabilistic: the method achieves synchronization only if the observed round-trip times between client and server are sufficiently short compared with the required accuracy.

A process **p** requests the time in a message $\mathbf{m_r}$ , and receives the time value **t** in a message $\mathbf{m_t}$ (t is inserted in $\mathbf{m_t}$ at the last possible point before transmission from S's computer).

Process **p** records the total round-trip time $\mathbf{T_{round}}$ taken to send the request $\mathbf{m_r}$ and receive the reply $\mathbf{m_t}$ . It can measure this time with reasonable accuracy if its rate of clock drift is small.

A simple estimate of the time to which **p** should set its clock is $\mathbf{t + T_{round} / 2}$ , which assumes that the elapsed time is split equally before and after **S** placed **t** in $\mathbf{m_t}$ .

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

The earliest point at which **S** could have placed the time in $\mathbf{m_t}$ was **min** after **p** dispatched $\mathbf{m_r}$ . The latest point at which it could have done this was **min** before $\mathbf{m_t}$ arrived at **p.**

The time by **S's** clock when the reply message arrives is therefore in the range $\mathbf{[t + min, t + T_{round} – min]}$ . The width of this range is $\mathbf{T_{round} – 2min}$ , so the accuracy is $\mathbf{\pm (T_{round} / 2 – min).}$

Variability can be dealt with to some extent by making several requests to *S* and taking the minimum value of $\mathbf{T_{round}}$ to give the most accurate estimate.

The greater the accuracy required, the smaller the probability of achieving it. This is because the most accurate results are those in which both messages are transmitted in a time close to **min** – an unlikely event in a busy network..

Cristian's method suffers from the problem associated with all services implemented by a single server: that the single time server might fail and thus render synchronization temporarily impossible.

Cristian suggested, for this reason, that time should be provided by a **group of synchronized time servers**, each with a receiver for UTC time signals.

## The Berkeley algorithm

In this 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).

The balance of probabilities is that this average cancels out the individual clocks' tendencies to run fast or slow. The accuracy of the protocol depends upon a nominal maximum round-trip time between the master and the slaves. The master eliminates any occasional readings associated with larger times than this maximum.

Instead of sending the updated current time back to the other computers – which would introduce further uncertainty due to the message transmission time – the master sends the amount by which each individual slave's clock requires adjustment. This can be a positive or negative value.

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

Should the master fail, then another can be elected to take over and function exactly as its predecessor.

## The Network Time Protocol

The Network Time Protocol (NTP) defines an architecture for a time service and a protocol to distribute time information over the Internet.

NTP's chief design aims and features are as follows:

i. To provide a service enabling clients across the Internet to be synchronized accurately to UTC.

ii. To provide a reliable service that can survive lengthy losses of connectivity thru redundant servers and redundant paths.

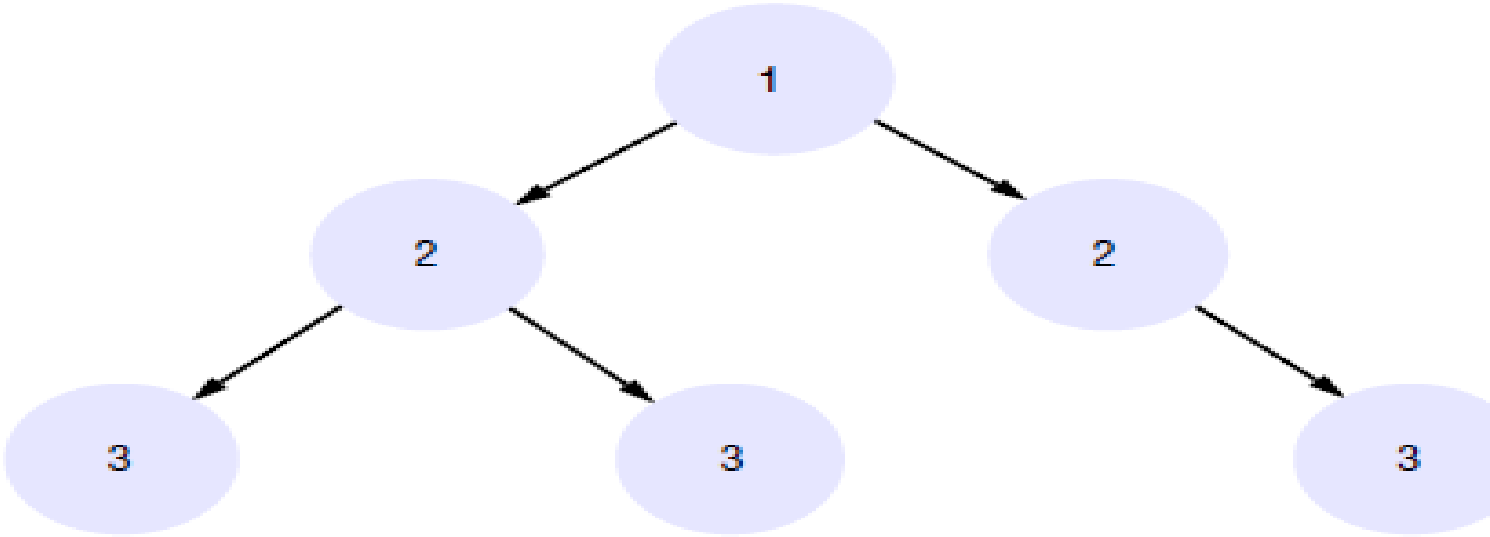
iii. To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers.

iv. To provide protection against interference with the time service, whether malicious or accidental using authentication services.

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 clock 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: they are at the root.

Stratum 2 servers are secondary servers that are synchronized directly with the primary servers; stratum 3 servers are synchronized with stratum 2 servers, and so on. The lowest-level (leaf) servers execute in users' workstations.




Arrows denote synchronization control, numbers denote strata.

Clocks of servers with high stratum numbers are less accurate than those with low stratum numbers, because of synchronization errors introduced at each level.

NTP also takes into account the total message round-trip delays to the root in assessing the quality of timekeeping data held by a particular server. The synchronization subnet can reconfigure as servers become unreachable or failures occur.

NTP servers synchronize with one another in one of three modes: multicast, procedure-call and symmetric mode.

**Multicast mode** 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.

This mode can achieve only relatively low accuracies, but ones that nonetheless are considered sufficient for many purpose.

In **procedure-call mode**, one server accepts requests from other computers, which it processes by replying with its timestamp (current clock reading).

This mode is suitable where higher accuracies are required than can be achieved with multicast, or where multicast is not supported in hardware.

**Symmetric mode** is intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet, where the highest accuracies are to be achieved.

A pair of servers 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.

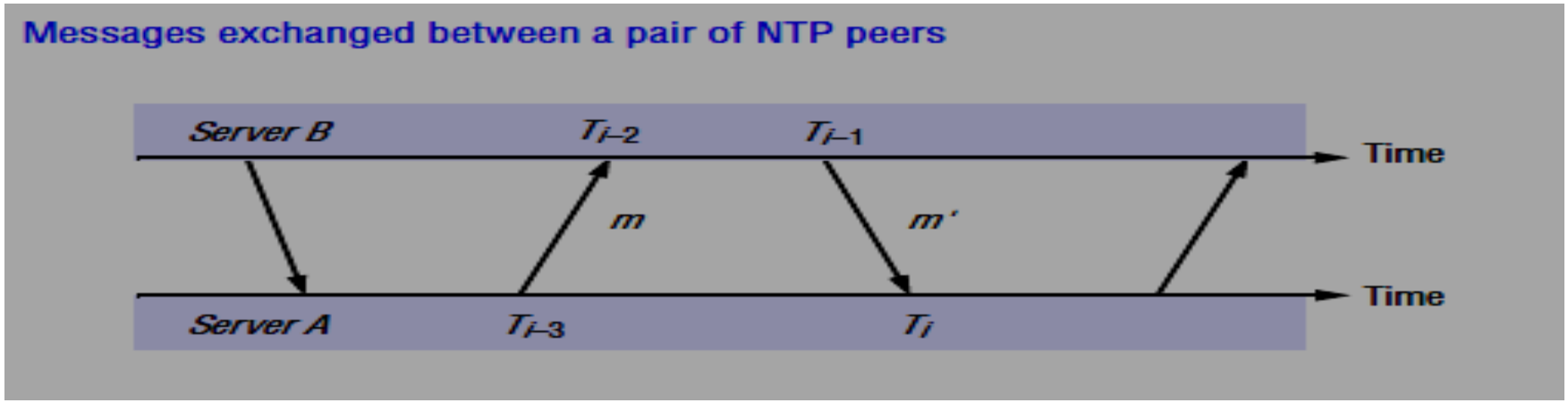
In all modes, messages are delivered unreliably, using the standard UDP Internet transport protocol. In procedure-call mode and symmetric mode, processes exchange pairs of messages.

Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted.

The recipient of the NTP message notes the local time when it receives the message

The four times $T_{i-3}$, $T_{i-2}$, $T_{i-1}$ and $T_i$ are shown in Fig for the messages m and m' sent between servers A and *B*. In symmetric mode, there can be a non-negligible delay between the arrival of one message and the dispatch of the next.

Also, messages may be lost, but the three timestamps carried by each message are nonetheless valid.



For each pair of messages sent between two servers the NTP calculates an offset $o_i$, which is an estimate of the actual offset between the two clocks, and a delay $d_i$, which is the total transmission time for the two messages.

If the true offset of the clock at B relative to that at A is o, and if the actual transmission times for m and m' are t and t', respectively, then we have:

$T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$

This leads to:

$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$

and:

$o = o_i + (t' - t)/2$ , where $o_i = (T_{i-2} - Ti - 3 + T_{i-1} - T_i) / 2$

Using the fact that t, t' > *0* , it can be shown that $o_i - d_{i/2} \leq o \leq o_i + d_{i/2}$ . Thus $o_i$ is an estimate of the offset, and $d_i$ is a measure of the accuracy of this estimate.

## Logical time and logical clocks

Ordering of events is based on two simple and intuitively obvious points:

i. If two events occurred at the same process $p_i$ (i = 1,2,… N) , then they occurred in the order in which $p_i$ observes them ( →i).

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

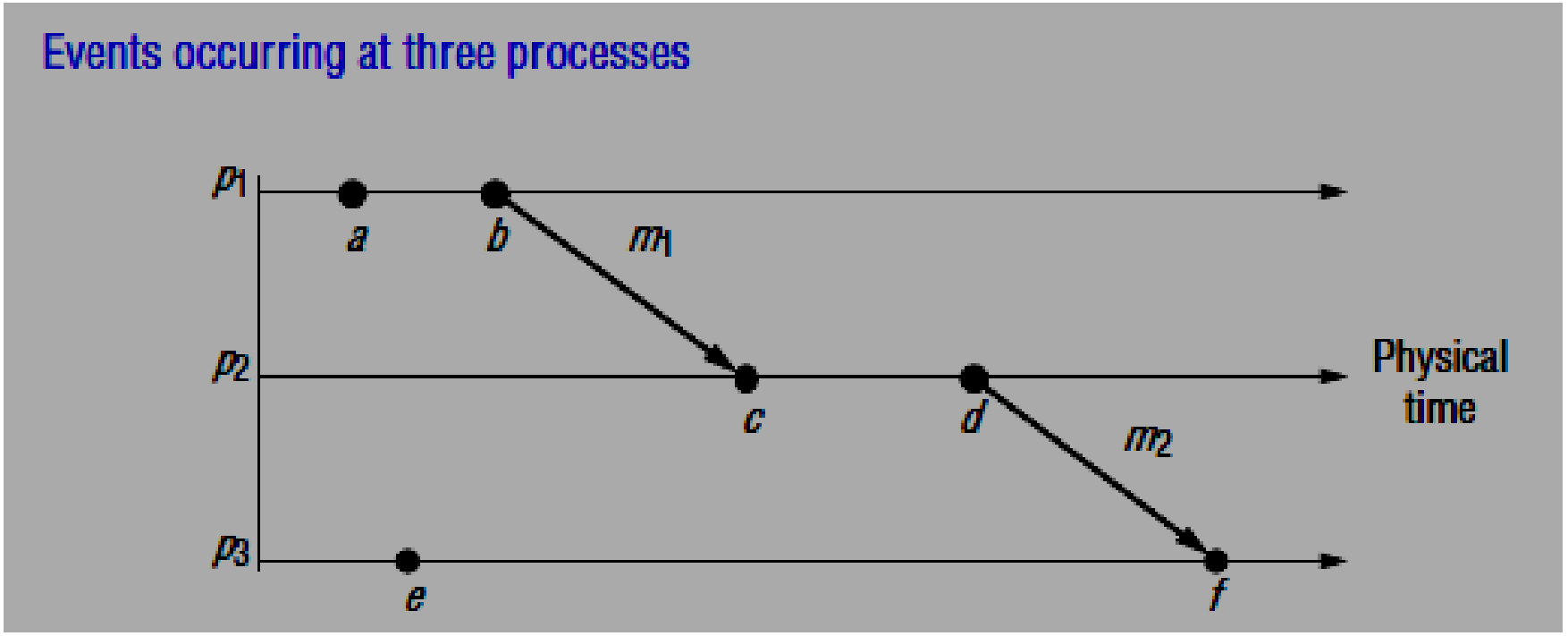
Lamport called the partial ordering obtained by generalizing these two relationships the **happened-before relation**. It is also sometimes known as the relation of **causal ordering or potential causal ordering**. We can define the happened-before relation, denoted by →, as follows:

HB1: If ∃(there exists) process $p_i$ : e → i e', then e → e'.

HB2: For any message m, **send(m)** → **receive(m)** – where **send(m)** is the event of sending the message, and **receive(m)** is the event of receiving it.

HB3: If e, e' and e'' are events such that e → e' and e' → e'' , then e → e'' .

Thus, if e and e' are events, and if e $\rightarrow$e' , then we can find a series of events $e_1$, $e_2$… $e_n$ occurring at one or more processes such that e = $e_1$ and e' = $e_n$ , and for i = 1, 2…N – 1 either HB1 or HB2 applies between $e_i$ and $e_{i+1.}$. That is, either they occur in succession at the same process, or there is a message m such that $e_i$ = **send(m)** and $e_{i+1}$ = **receive(m).**



The relation → is illustrated for the case of three processes, p1 , p2 and p3 , in Fig. It can be seen that a →b, and b →c, and similarly c → d. Since these events are the sending and reception of message m1 , and similarly d → f. Combining these relations, we may also say that, for example, a →f.

It can also be seen from Fig that not all events are related by the relation →. We say that events such as a and e that are not ordered by →, are not ordered are **concurrent** and write this as a || e.

Another point to note is that if the **happened-before** relation holds between two events, then the first might or might not actually have caused the second. A process might, for example, receive a message and subsequently issue another message, but one that it issues every five minutes anyway and that bears no specific relation to the first message. No actual causality has been involved, but the relation → would order these events.

## Logical clocks

Lamport invented a e 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, whose value bears no particular relationship to any physical clock.

Each process $p_i$ keeps its own logical clock, $L_i$ .We denote the timestamp of event **e** at $p_i$ by $L_i(e)$ , and by L(e) we denote the timestamp of event **e** at whatever process it occurred at.

To capture the **happened-before** relation $\rightarrow$ , processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

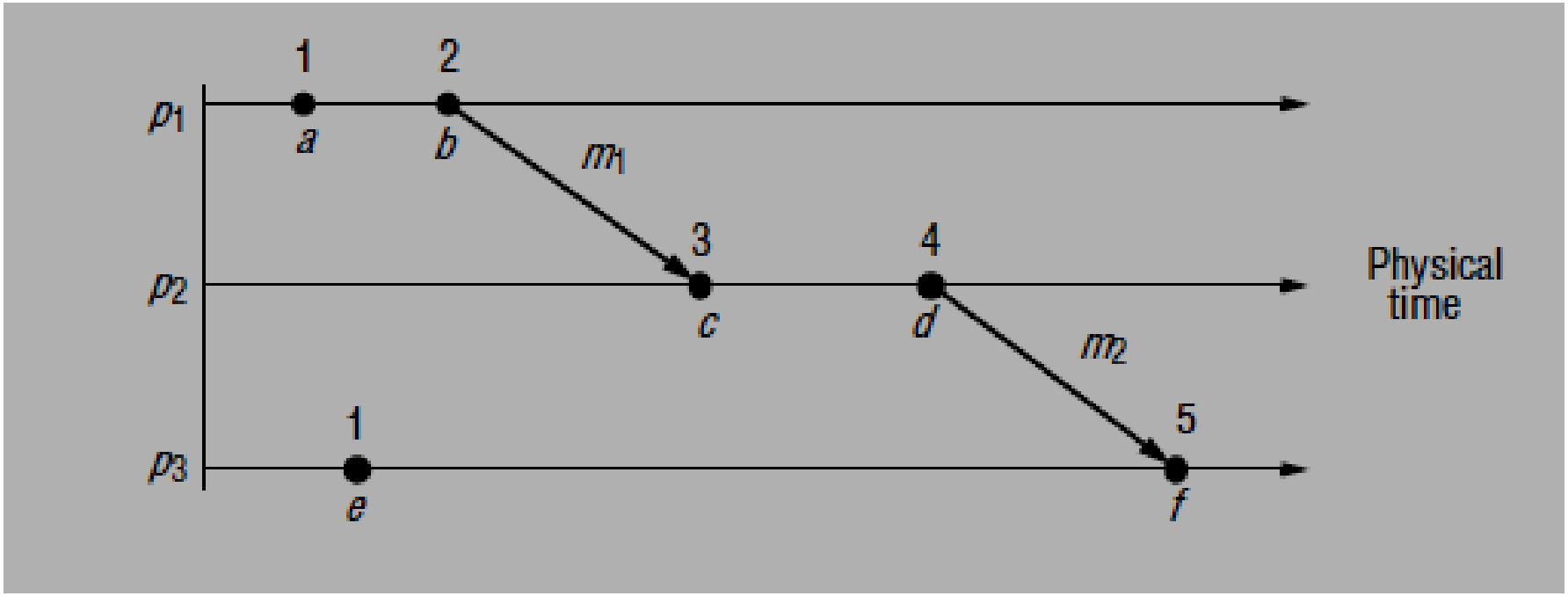
LC1: $L_i$ is incremented before each event is issued at process $p_i$: $L_i := L_i + 1$.

LC2: (a) When a process $p_i$ sends a message **m**, it piggybacks on **m** the value $t = L_i$ .

(b) On receiving (**m, t**), a process $p_i$ computes $L_j := \max(L_j , t)$ and then applies LC1 before time stamping the event **receive(m)**.

It can easily be shown, by induction on the length of any sequence of events relating two events e and e' , that e → e' => L(e) < L(e') . Note that the converse is not true.

If L(e) < L(e') , then we cannot infer that e → e' . In Fig we illustrate the use of logical clocks. Each of the processes p1 , p2 and p3 has its logical clock initialized to 0. The clock values given are those immediately after the event to which they are adjacent. Note that, for example, 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. However, we can create a total order on the set of events by taking into account the identifiers of the processes at which events occur.

If e is an event occurring at $p_i$ with local timestamp $T_i$ , and e' is an event occurring at $p_j$ with local timestamp $T_j$ , we define the global logical time stamps for these events to be $(T_i , i)$ and $(T_j, j)$ , respectively. And we define $(T_i , i ) < (T_j, j)$ **if and only if** either $T_i < T_j$ , or $T_i = T_j$ and $i < j$ .

## Vector clocks

Vector clocks were developed to overcome the shortcoming of Lamport's clocks: the fact that from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$.

A vector clock for a system of N processes is an array of N integers. Each process keeps its own vector clock, Vi , which it uses to timestamp local events. Processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:

VC1: Initially, $V_i[j] = 0$ , for i, j = 1, 2…N .

VC2: Just before $p_i$ timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

VC3: $p_i$ includes the value $t = V_i$ in every message it sends.

VC4: When $p_i$ receives a timestamp t in a message, it sets $V_i[j] := \max(V_i[j], t[j])$, 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 $V_i$ , $V_i[i]$ is the number of events that $p_i$ has time stamped, and $V_i[j]$ ($j \neq i$) is the number of events that have occurred at $p_j$ that have potentially affected $p_i$ .

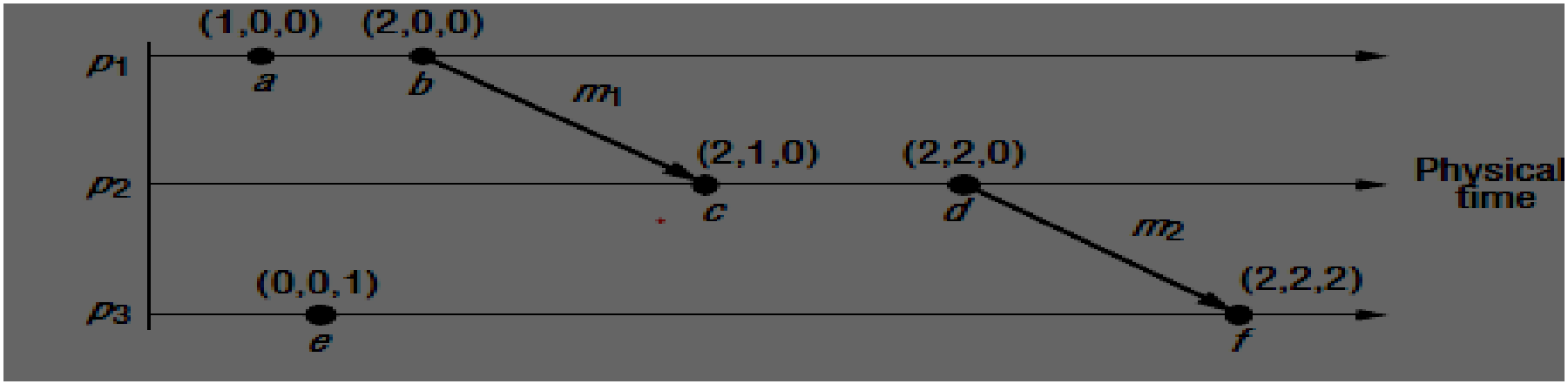


Figure shows the vector timestamps of the events. It can be seen, for example, that V(a) <V(f) , which reflects the fact that a → f. Similarly, we can tell when two events are concurrent by comparing their timestamps.

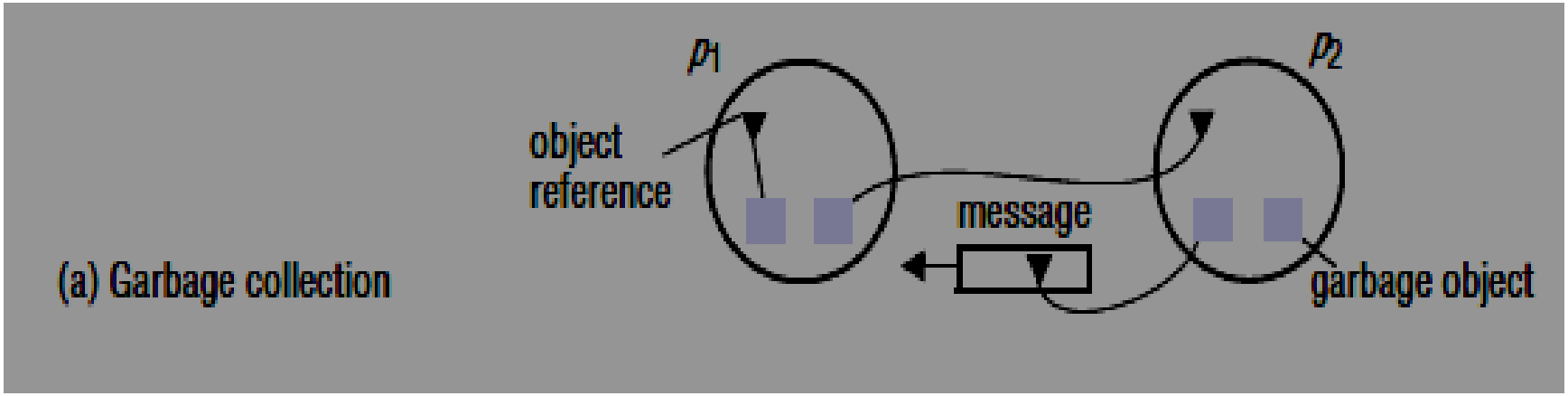
Vector timestamps have the disadvantage, compared with Lamport timestamps, of taking up an amount of storage and message payload that is proportional to N, the number of processes.

## Global states

i. **Distributed garbage collection:** If an object is no longer referenced any where, it is considered to be garbage and the memory taken up by that object can be reclaimed.

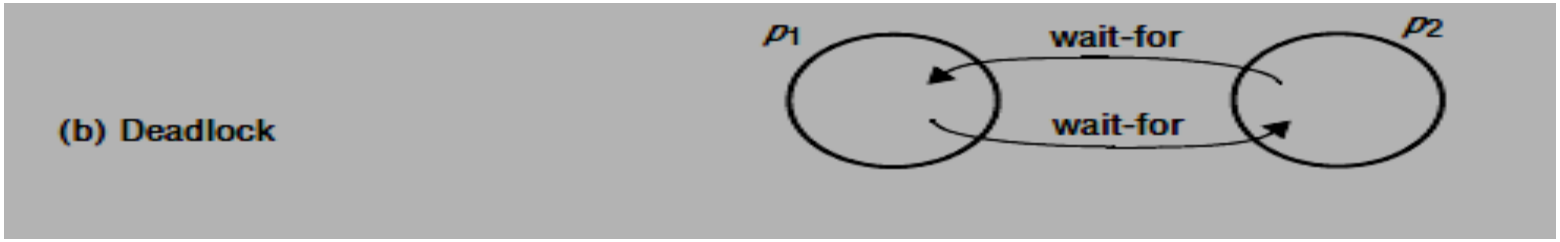
In Figure, process p1 has two objects that both have references – one has a reference within p1 itself, and p2 has a reference to the other. Process p2 has one garbage object. It also has an object that has a reference to it in a message that is in transit between the processes.

This shows that when we consider properties of a system, we must include the state of communication channels as well as the state of the processes.




ii. **Distributed deadlock detection:** A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this 'waits-for' relationship.

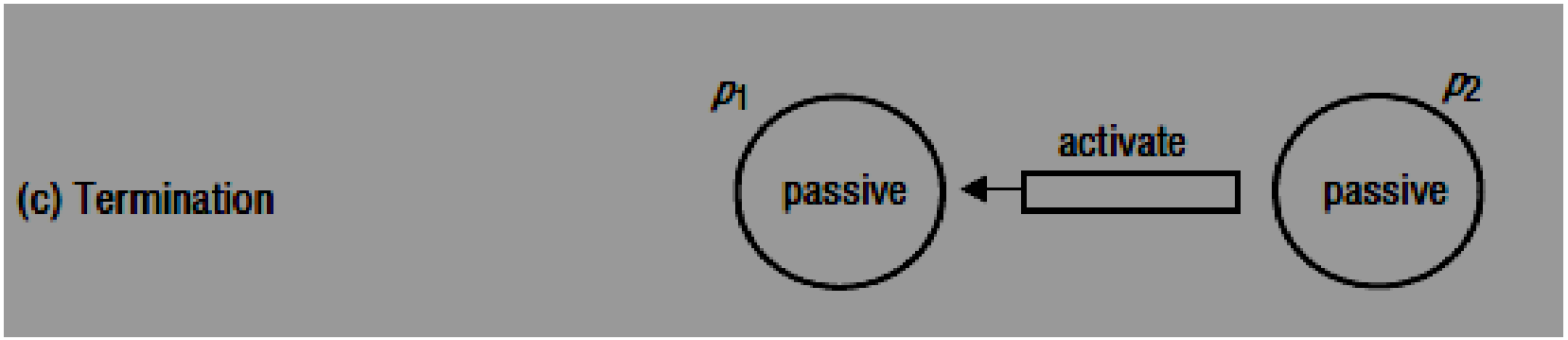
In the Fig, processes p1 and p2 are each waiting for a message from the other, so this system will never make progress.




iii. **Distributed termination detection:** The problem here is how to detect that a distributed algorithm has terminated. Consider a distributed algorithm executed by two processes p1 and p2 , each of which may request values from the other.

A process is either active or passive – a passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other. Suppose if p1 is passive and that p2 is passive.

To see that the algorithm has not terminated, consider the following scenario: when p1 was tested for passivity, a message was on its way from p2 , which became passive immediately after sending it. On receipt of the message, p1 became active again – after it was found it to be passive. Thus the algorithm had not terminated.




**Similarity and difference between deadlock and termination:**

i. First, a deadlock may affect only a subset of the processes in a system, whereas all processes must have terminated.

ii. Second, process passivity is not the same as waiting in a deadlock cycle: a deadlocked process is attempting to perform a further action, for which another process waits; a passive process is not engaged in any activity.

**iv. Distributed debugging:**

Distributed systems are complex to debug, and care needs to be taken in establishing what occurred during the execution. For example, an application in which each process pi contains a variable xi ( i = 1, 2…N ).

The variables change as the program executes, but they are required always to be within a value $\delta$ of one another. Because of some bug, under certain circumstances $|x_i - x_j| > \delta$ for some i and j. The problem is that, this relationship must be evaluated for **values of the variables that occur at the same time.**

## Global states and consistent cuts

The successive states of an individual process is observable, but ascertaining a global state of the system i.e., the state of the collection of processes – is much harder to address due to the absence of global time. All processes could not perfectly synchronize their clocks, and hence prediction of global state of system is not possible.

## The 'snapshot' algorithm of Chandy and Lamport

The goal of the algorithm is to record a set of process and channel states (a 'snapshot') for a set of processes $p_i$ ( $i = 1,2\ldots N$ ) such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.

The algorithm assumes that:

•Neither channels nor processes fail – communication is reliable so that every message sent is eventually received intact, exactly once.

• Channels are unidirectional and provide FIFO-ordered message delivery.

•The graph of processes and channels is strongly connected (there is a path between any two processes).

• Any process may initiate a global snapshot at any time.

•The processes may continue their execution and send and receive normal messages while the snapshot takes place.

For each process $p_i$ , let the **incoming channels** be those at $p_i$ over which other processes send it messages; similarly, the **outgoing channels** of $p_i$ are those on which it sends messages to other processes.

Each process records its state and also, for each incoming channel, a set of messages sent to it. The process records, for each channel, any messages that arrived after it recorded its state and before the sender recorded its own state.

The algorithm proceeds through use of special **marker messages**, which are distinct from any other messages the processes send and which the processes may send and receive while they proceed with their normal execution.

The marker has a dual role:

- as a prompt for the receiver to save its own state, if it has not already done so;
- 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 messages.

The marker receiving rule obligates a process that has not recorded its state to do so. In that case, this is the first marker that it has received. It notes which messages subsequently arrive on the other incoming channels.

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

## Chandy and Lamport's 'snapshot' algorithm

***Marker receiving rule for process*** $p_i$

On receipt of a *marker* message at $p_i$ over channel $c$:

    *if* ($p_i$ 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*
        $p_i$ records the state of $c$ as the set of messages it has received over $c$
        since it saved its state.
    *end if*

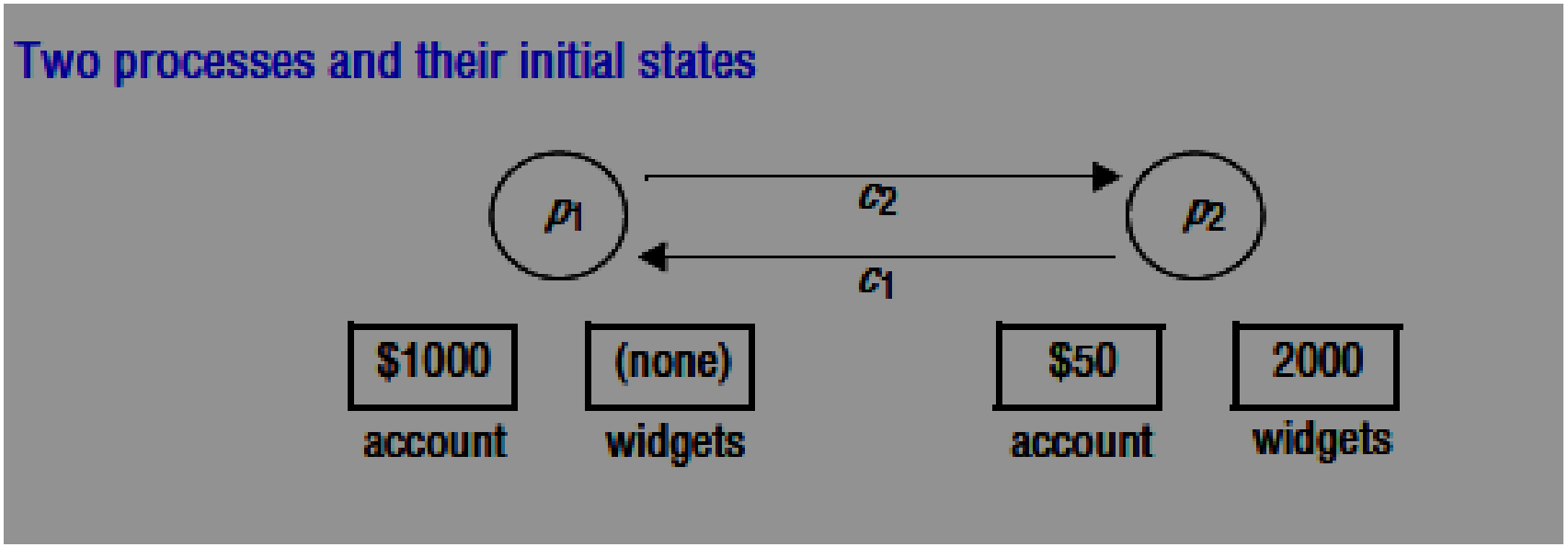
***Marker sending rule for process*** $p_i$

After $p_i$ has recorded its state, for each outgoing channel $c$:

    $p_i$ sends one marker message over $c$
    (before it sends any other message over $c$).

The algorithm is illustrated for a system of two processes, $p_1$ and $p_2$ , connected by two unidirectional channels, $c_1$ and $c_2$ . The two processes trade in 'widgets'. Process $p_1$ sends orders for widgets over $c_2$ to $p_2$ , enclosing payment at the rate of $10 per widget. Some time later, process $p_2$ sends widgets along channel $c_1$ to $p_1$ .

The processes have the initial states shown in Fig. Process $p_2$ has received an order for five widgets, which it will shortly dispatch to $p_1$.



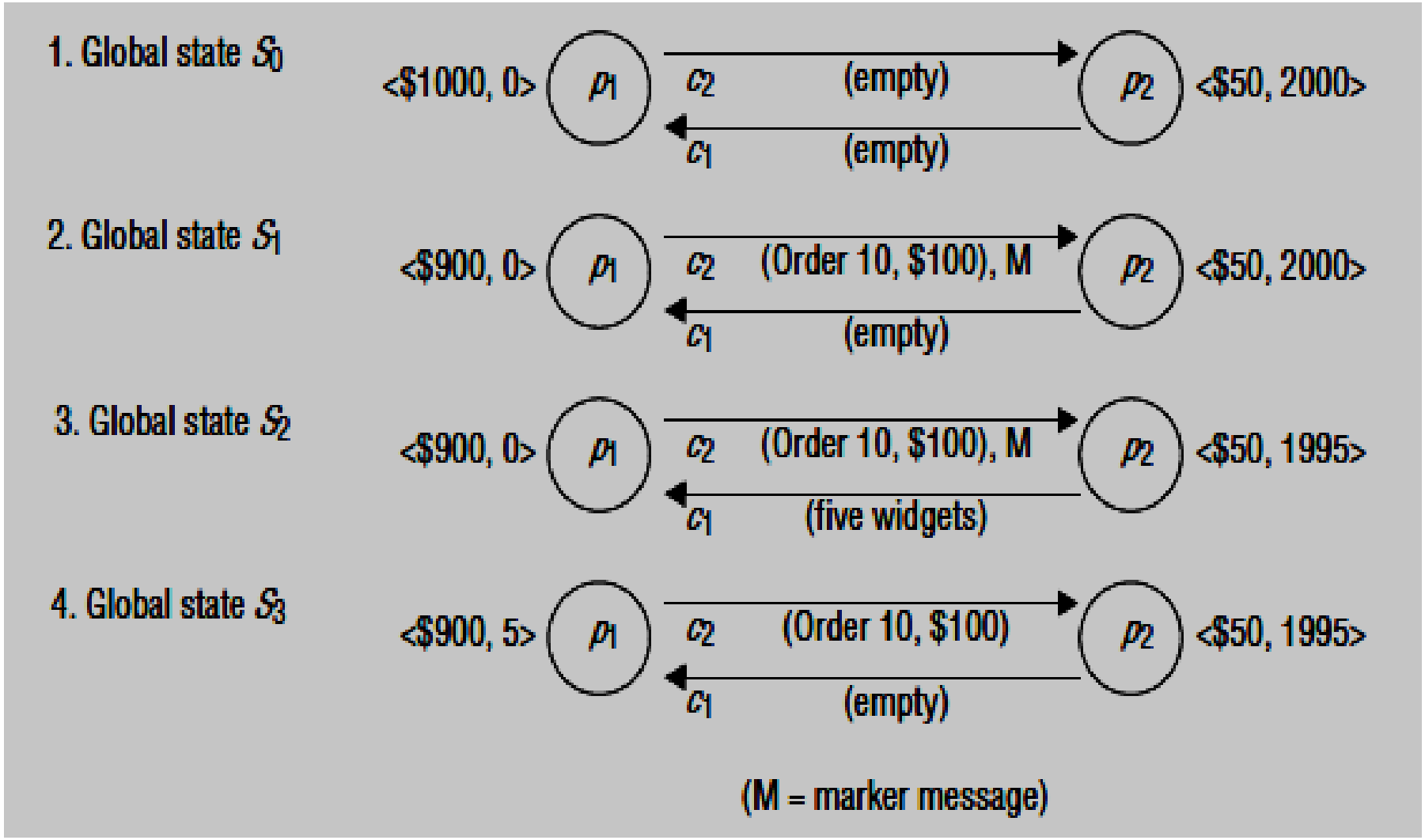
The below Fig shows an execution of the system while the state is recorded. Process p1 records its state in the actual global state S0 , when the state of p1 is <$1000, 0>. Process p1 then emits a marker message over its outgoing channel c2 before it sends the next application-level message: (Order 10, $100), over channel c2 .

The system enters actual global state S1. Before p2 receives the marker, it emits an application message (five widgets) over c1 in response to p1 ’s previous order, yielding a new actual global state S2 .

Now process p1 receives p2 ’s message (five widgets), and p2 receives the marker. P2 records its state as <$50, 1995> and that of channel c2 as the sequence. It sends a marker message over c1 .

When process p1 receives p2 ’s marker message, it records the state of channel c1 as the single message (five widgets) that it received after it first recorded its state. The final actual global state is S3 .

The final recorded state is $p1$ : <$1000, 0>; $p2$ : <$50, 1995>; $c1$ : <(five widgets)>; $c2$ : < >. Note that this state differs from all the global states through which the system actually passed.



## Termination of the snapshot algorithm

A process that has received a marker message records its state within a finite time and sends marker messages over each outgoing channel within a finite time.

If there is a path of communication channels, and processes from a process pi to a process pj ($j \neq i$), then pj will record its state a finite time after pi recorded its state. Therefore, all processes will have recorded their states and the states of incoming channels a finite time, after some process initially records its state.

# COORDINATION AND AGREEMENT

## Introduction

In a distributed system, a set of processes coordinate their actions or agree on one or more values.

In an asynchronous system no timing assumptions are made. In a synchronous system, there are bounds on the maximum message transmission delay, on the time taken to execute each step of a process, and on clock drift rates. The synchronous assumptions allow us to use timeouts to detect process crashes.

Even under surprisingly benign(non threatening) failure conditions, it is impossible to guarantee in an asynchronous system that a collection of processes can agree on a shared value.

## Failure assumptions and failure detectors

Assumptions are:

i. Each pair of processes is connected by reliable channels. Network failures are masked by retransmitting missing or corrupted messages.

ii. An individual process's failure cannot be a threat to the other processes' ability to communicate. This means that none of the processes depends upon another to forward messages.

iii. Hardware redundancy is present in synchronous systems that guarantees message is not only eventually delivered but within specified time bounds.

iv. In any particular interval of time, communication between some processes may succeed while communication between others is delayed.

v. Network partitioning is a state where intra-pair communication among pair of nodes is possible over their respective networks; but inter-pair communication is not possible

vi. In **asymmetric connectivity**, communication is possible from process p to process q, but not vice versa. In **intransitive connectivity,** communication is possible from p to q and from q to r, but p cannot communicate directly with r.

vii. Eventually any failed link or router will be repaired or circumvented. Nevertheless, the processes may not all be able to communicate at the same time.

viii. A **correct** process is one that exhibits no failures at any point in the execution under consideration. Correctness applies to the whole execution, not just to a part of it. A process that suffers a crash failure is ‘non-failed ’before that point, not ‘correct’ after that point.

A **failure detector** is a service that processes queries about whether a particular process has failed. It is often implemented by an object local to each process that runs a failure-detection algorithm in conjunction with its counterparts at other processes. The object local to each process is called a **local failure detector**.

An unreliable failure detector may produce one of two values when given the identity of a process: **Unsuspected** or **Suspected**. A result of **Unsuspected** signifies that the detector has recently received evidence suggesting that the process has not failed;

A result of **Suspected** signifies that the failure detector has some indication that the process may have failed.

A **reliable failure detector** is one that is always accurate in detecting a process's failure. A result of **Unsuspected means** as before, can only be a hint. A result of **Failed** means that the detector has determined that the process has crashed.

An unreliable failure detector using the following algorithm:
Each process p sends a '**p is here**' message to every other process, for every **T** seconds with an estimate of maximum message transmission time of **D** seconds. If the local failure detector at process q does not receive a '**p is here**' message within **T + D** seconds of the last one, then it reports to q that p is **Suspected**. However, if it subsequently receives a '**p is here**' message, then it reports to q that p is **OK**.

In real time distributed systems, if a small value is chosen for T and D , then the failure detector is likely to suspect non-crashed processes many times, and much bandwidth will be taken up with '**p is here**' messages.

On choosing a large total timeout value, then crashed processes will often be reported as **Unsuspected**. A practical solution to this problem is to use timeout values that reflect the observed network delay conditions.

In a synchronous system, failure detector can be made into a reliable one by choosing D to be an absolute bound on message transmission time; the absence of a '**p is here**' message within **T + D** seconds entitles the local failure detector to conclude that p has crashed.

Unreliable failure detectors may suspect a process that has not failed (they may be inaccurate), and they may not suspect a process that has in fact failed (they may be incomplete). Reliable failure detectors, on the other hand, require that the system is synchronous.

## Distributed mutual exclusion

Distributed processes often need to coordinate their activities. Mutual exclusion is required to prevent interference and ensure consistency when accessing the shared resources. This is the critical section problem, familiar in the domain of operating system.

In a distributed system, however, a solution to distributed mutual exclusion is based solely on message passing.

## Algorithms for mutual exclusion

Consider a system of N processes $\mathbf{p_i}$ , i = 1, 2,.. N , that do not share variables. The processes access common resources, but they do so in a critical section, which is only one.

Assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.

The application-level protocol for executing a critical section is as follows:

enter() // enter critical section – block if necessary

resourceAccesses() // access shared resources in critical section

exit() // leave critical section – other processes may now enter.

Our essential requirements for mutual exclusion are as follows:

ME1: (safety) At most one process may execute in the critical section (CS) at a time.

ME2: (liveness) Requests to enter and exit the critical section eventually succeed.

Condition ME2 implies freedom from both **deadlock** and **starvation**. A deadlock would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section, by virtue of their mutual interdependence.

Starvation is the indefinite postponement of entry for a process that has requested it. The absence of starvation is a **fairness** condition.

ME3: (→ ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

If a solution grants entry to the critical section in happened-before order, and if all requests are related by happened-before, then it is not possible for a process to enter the critical section more than once while another waits to enter. This ordering also allows processes to coordinate their accesses to the critical section.

A multi-threaded process may continue with other processing while a thread waits to be granted entry to a critical section. ME3 specifies that the first process be granted access before the second.

**Criteria for performance evaluation:**

i. The bandwidth consumed, which is proportional to the number of messages sent in each entry and exit operation;

ii. The client delay incurred by a process at each entry and exit operation;

iii. The algorithm's effect upon the **throughput** of the system. This is the rate at which the collection of processes as a whole can access the critical section, given that some communication is necessary between successive processes.

This is measured using the **synchronization delay** between one process exiting the critical section and the next process entering it; the throughput is greater when the synchronization delay is shorter.

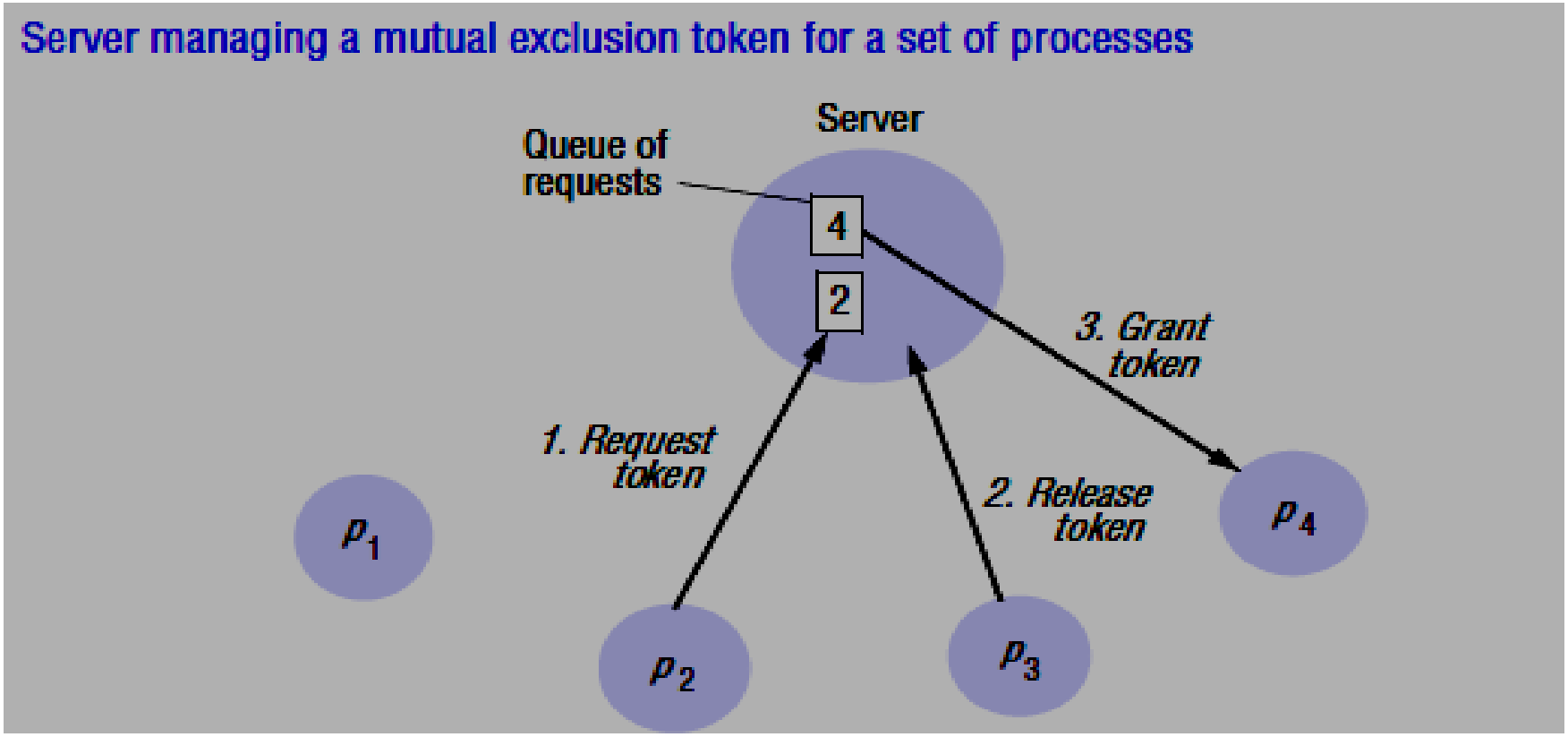
The implementation of resource accesses is not taken into account. However, it is assumed that the client processes are well behaved and spend a finite time accessing resources within their critical sections.

## The central server algorithm

A server is employed 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. Conceptually, the reply constitutes a token signifying permission to enter the critical section.

If the token is free, the server replies immediately, granting the token. Else, the server does not reply, but queues the request. When a process exits the critical section, it sends a message to the server, giving it back the token.

The server chooses the oldest entry in the queue, removes it and replies to the corresponding process. The chosen process then holds the token.



We now evaluate the performance of this algorithm. Entering the critical section – even when no process currently occupies it – takes two messages (a **request** followed by a **grant**) and delays the requesting process by the time required for this round-trip.

Exiting the critical section takes one **release** message. Assuming asynchronous message passing, this does not delay the exiting process.

The server may become a performance bottleneck for the system as a whole. The synchronization delay is the time taken for a round-trip: a **release** message to the server, followed by a **grant** message to the next process to enter the critical section.

## A ring-based algorithm

One of the simplest ways to arrange mutual exclusion between the N processes without requiring an additional process is to arrange them in a logical ring.

This requires only that each process $\mathbf{p_i}$ has a communication channel to the next process in the ring, $\mathbf{p_{(i + 1) \bmod N}}$.

Exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction – clockwise, say – around the ring.

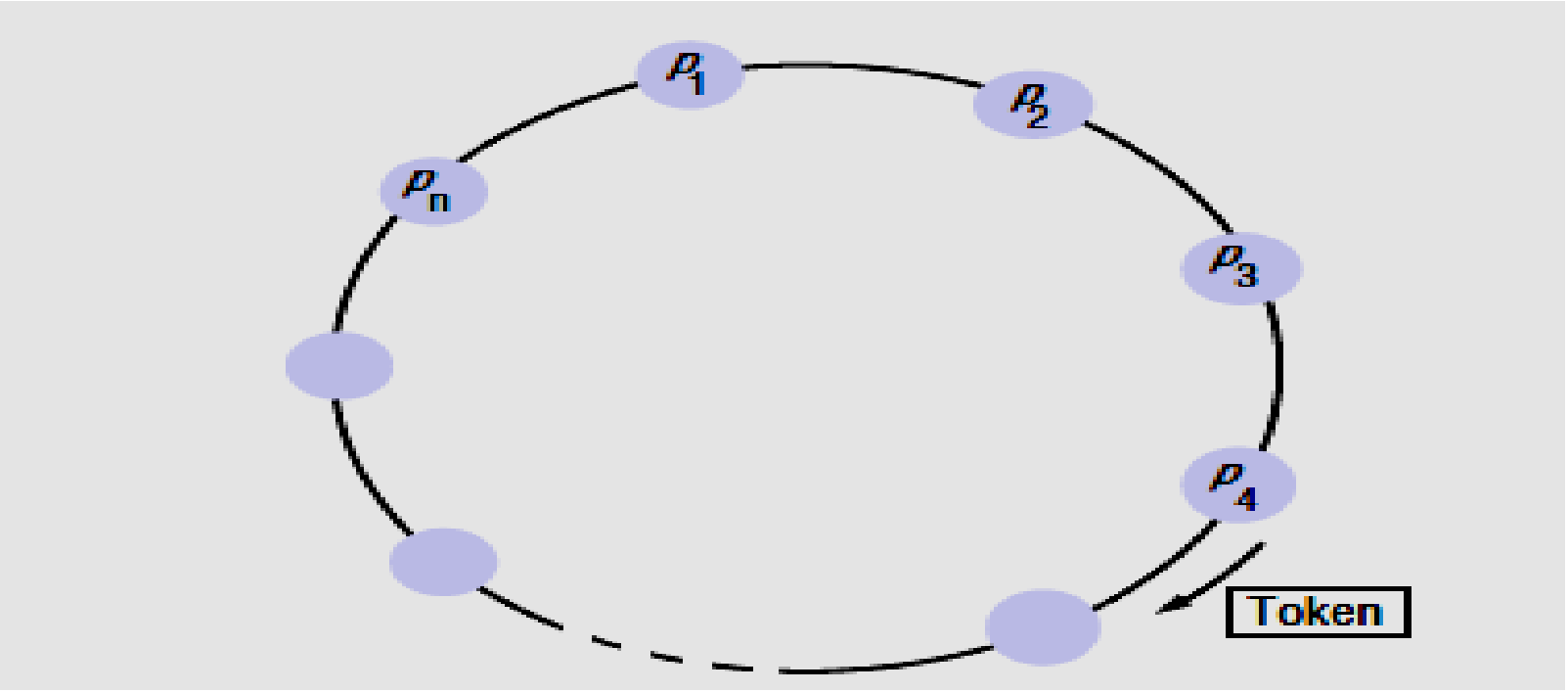
If a process does not require to enter the critical section, forwards the token to its neighbour. A process that requires the token waits until it receives it, but retains it. To exit the critical section, the process sends the token on to its neighbour.

The arrangement of processes is shown in Fig. It is straightforward to verify that the conditions ME1 and ME2 are met by this algorithm, but that the token is not necessarily obtained in happened-before order.

This algorithm continuously consumes network bandwidth (except when a process is inside the critical section): the processes send messages around the ring even when no process requires entry to the critical section.

The delay experienced by a process requesting entry to the critical section is between 0 messages (when it has just received the token) and N messages (when it has just passed on the token).

To exit the critical section requires only one message. The synchronization delay between one process's exit from the critical section and the next process's entry is anywhere from 1 to N message transmissions.



## An algorithm using multicast and logical clocks

Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message. The conditions under which a process replies to a request are designed to ensure that conditions ME1–ME3 are met.

The processes $p_1$, $p_2$.. $p_N$ bear distinct numeric identifiers. They are assumed to possess communication channels to one another, and each process $p_i$ keeps a Lamport's clock. Messages requesting entry are of the form <T, $p_i$ >, where T is the sender's timestamp and $p_i$ is the sender's identifier.

Each process records its state of being outside the critical section (RELEASED), wanting entry (WANTED) or being in the critical section (HELD) in a variable state. The protocol is given in Fig.

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

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

If two or more processes request entry at the same time, then whichever process's request bears the lowest timestamp will be the first to collect N – 1 replies, granting it entry next. If the requests bear equal Lamport timestamps, the requests are ordered according to the processes' corresponding identifiers.

When a process requests entry, it defers processing requests from other processes until its own request has been sent and it has recorded the timestamp T of the request. This is so that processes make consistent decisions when processing requests.

This algorithm achieves the safety property ME1. If it were possible for two processes $p_i$ and $p_j$ ( $i \neq j$ ) to enter the critical section at the same time, then both of those processes would have to have replied to the other. But since the pairs $< T_i , p_i >$ are totally ordered, this is impossible.

## Ricart and Agrawala's algorithm

*On initialization*

*state* := RELEASED;

*To enter the section*

*state* := WANTED;
Multicast *request* to all processes;
*T* := request's timestamp;

*Request processing deferred here*

*Wait until* (number of replies received = $(N-1)$);
*state* := HELD;

*On receipt of a request* $\langle T_i, p_i \rangle$ *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*

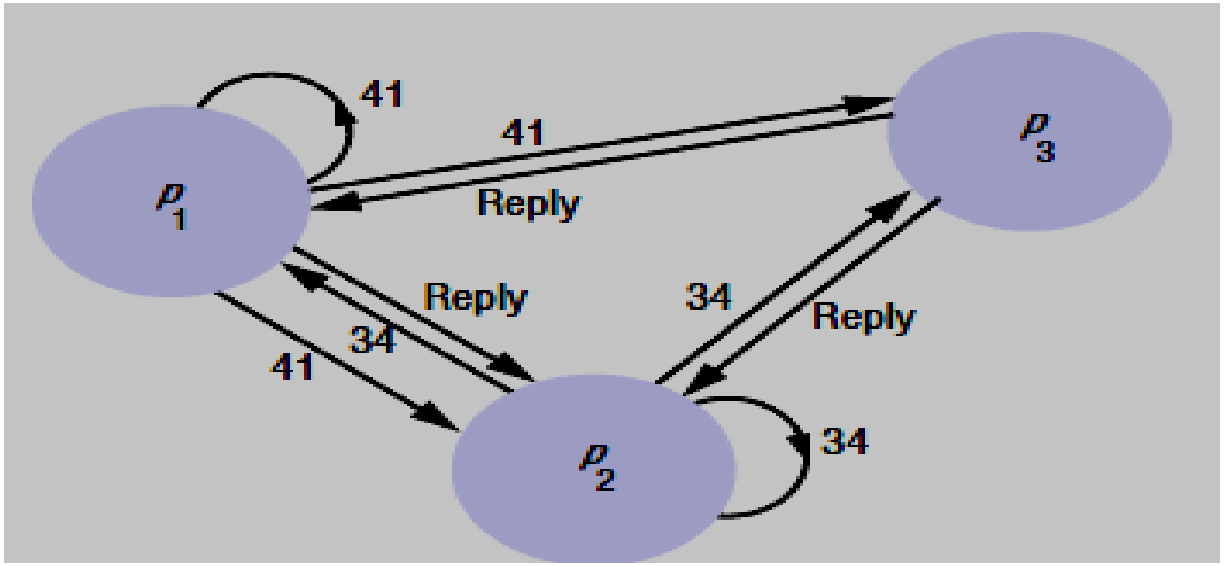
*To exit the critical section*

*state* := RELEASED;
reply to any queued requests;

Consider three processes, p1 , p2 and p3 , shown in Fig. Assume that p1 and p2 request entry concurrently and p3 is not interested in entering the critical section. The timestamp of p1 and p2 is 41 and 34 respectively.

When p3 receives their requests, it replies immediately. When p2 receives p1's request, it finds that its own request has the lower timestamp and so does not reply, holding p1 off.

However, p1 finds that p2 's request has a lower timestamp and so replies immediately. On receiving this second reply, p2 can enter the critical section. When p2 exits 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. Or, if there is hardware support for multicast, only one message is required for the request; the total is then N messages. It is thus a more expensive algorithm, in terms of bandwidth consumption.

The advantage of this algorithm is that its synchronization delay is only one message transmission time.

**Maekawa's voting algorithm**

In order for a process to enter a critical section, it is not necessary for all of its peers to grant it access. Processes need only obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.

Processes need to vote for one another to enter the critical section. A 'candidate' process must collect sufficient votes to enter. Processes in the intersection of two sets of voters ensure the safety property ME1, that at most one process can enter the critical section, by casting their votes for only one candidate.

## 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

An algorithm for choosing a unique process to play a particular role is called an election algorithm.

Each process $p_i$ ( i = 1, 2.. N ) has a variable **elected$_i$** , which will contain the identifier of the elected process. When the process first becomes a participant in an election it sets this variable to the special value ‘ $\perp$ ’ to denote that it is not yet defined.

Our requirements are that, during any particular run of the algorithm:

E1: (safety) A participant process $p_i$ has **elected$_i$** = $\perp$ or **elected$_i$** = P, where P is chosen as the non-crashed process at the end of the run with the largest identifier.

E2: (liveness) All processes $p_i$ participate and eventually either set **elected$_i$** $\neq$ $\perp$ or crash.

There may be processes $p_j$ that are not yet participants, which record in $\mathbf{elected_j}$ the identifier of the previous elected process.

The performance of an election algorithm is measured by its total network bandwidth utilization (which is proportional to the total number of messages sent), and by the turnaround time for the algorithm: the number of serialized message transmission times between the initiation and termination of a single run.

## A ring-based election algorithm

The algorithm is suitable for a collection of processes arranged in a logical ring. Each process $p_i$ has a communication channel to the next process in the ring, $\mathbf{p_{(i + 1) mod N}}$, and all messages are sent clockwise around the ring.

We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect a single process called the **coordinator,** which is the process with the largest identifier.

Initially, every process is marked as a **non-participant** in an election. Any process can begin an election. It proceeds by marking itself as a **participant**, placing its identifier in an election message and sending it to its clockwise neighbour.

When a process receives an election message, it compares the identifier in the message with its own. If the arrived identifier is greater, then it forwards the message to its neighbour. If the arrived identifier is smaller and the receiver is not a **participant**, then it substitutes its own identifier in the message and forwards it;

It does not forward the message if it is already a **participant**. On forwarding an election message in any case, the process marks itself as a **participant**.

If, however, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the **coordinator**. The coordinator marks itself as a **non-participant** once more and sends an elected message to its neighbour, announcing its election and enclosing its identity.

When a process $p_i$ receives an elected message, it marks itself as a **non-participant**, sets its variable **electedi** to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbour.

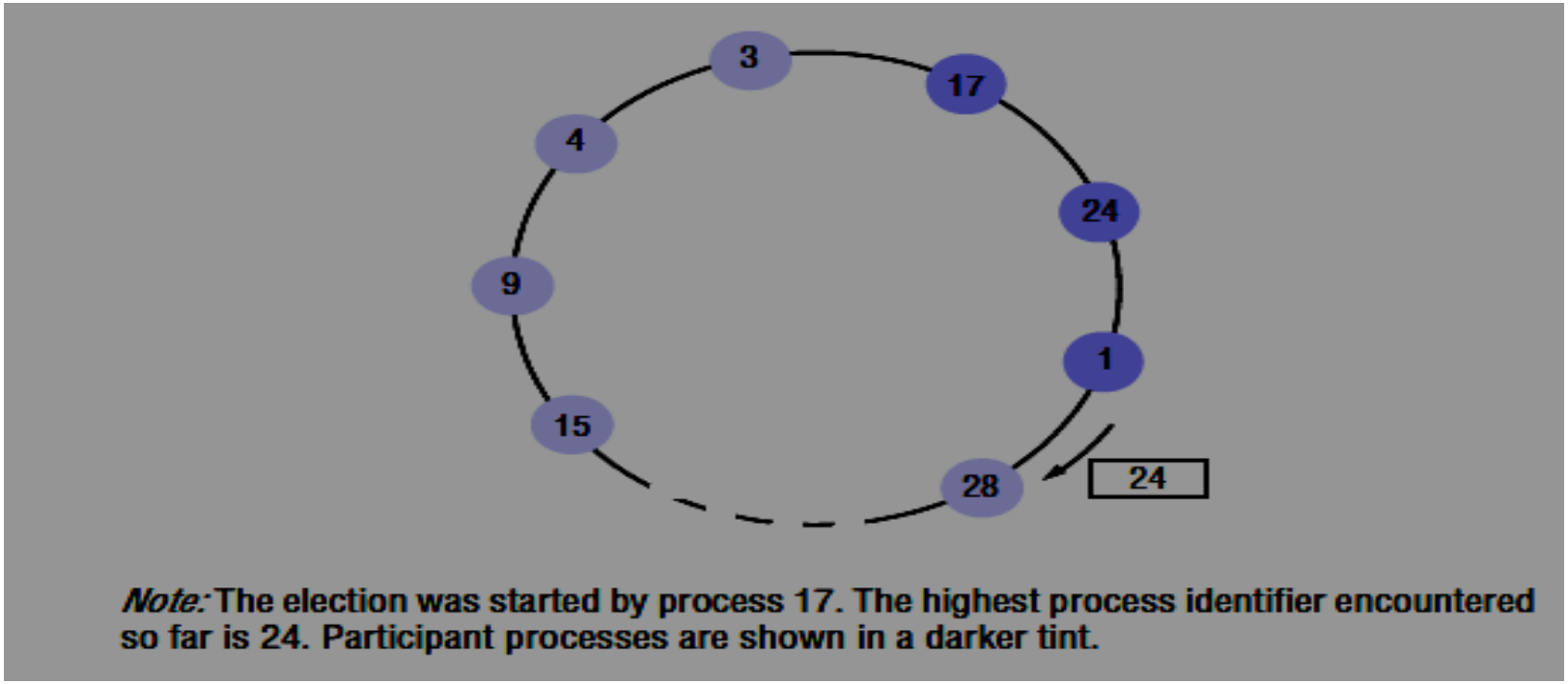
It is easy to see that condition E1 is met. All identifiers are compared, since a process must receive its own identifier back before sending an **elected** message. For any two processes, the one with the larger identifier will not pass on the other's identifier. It is therefore impossible that both should receive their own identifier back.

Condition E2 follows immediately from the guaranteed traversals of the ring (there are no failures).

The **non-participant** and **participant** states are used so that duplicate messages arising when two processes start an election at the same time are extinguished as soon as possible, and always before the 'winning' election result has been announced.

If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier. A total of N – 1 messages are then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages.

The **elected** message is then sent N times, making 3N – 1 messages in all. The turnaround time is also 3N – 1 , since these messages are sent sequentially.




*Note:* The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a darker tint.

## The bully algorithm

The bully algorithm allows processes to crash during an election, although it assumes that message delivery between processes is reliable.

The algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure. Also each process knows which processes have higher identifiers, and that it can communicate with all such processes.

There are three types of message in this algorithm: **an election message** is sent to announce an election; an **answer message** is sent in response to an election message and **a coordinator message** is sent to announce the identity of the elected process – the new 'coordinator'.

A process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently.

There is a maximum message transmission delay, $T_{trans}$ , and a maximum delay for processing a message $T_{process}$ . Therefore, time $T = 2T_{trans} + T_{process}$ that is an upper bound on the time that can elapse between sending a message to another process and receiving a response.

If no response arrives within time T, then the local failure detector can report that the intended recipient of the request has failed.

The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a **coordinator message** to all processes with lower identifiers.

On the other hand, a process with a lower identifier can begin an election by sending an election message to those processes that have a higher identifier and awaiting answer messages in response. If none arrives within time T, the process considers itself the coordinator and sends a **coordinator message** to all processes with lower identifiers announcing this.

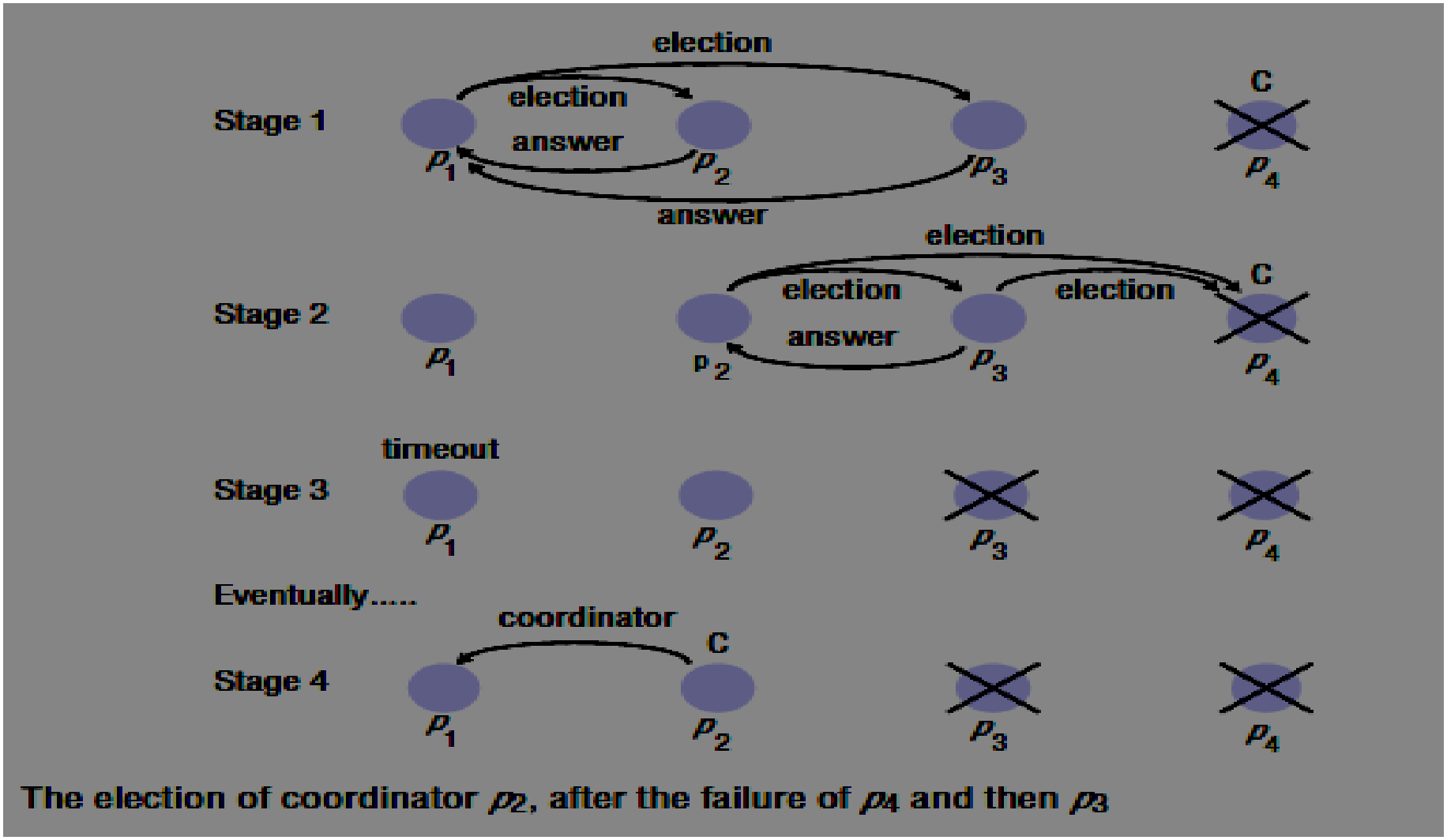
Otherwise, the process waits a further period T' for a coordinator message to arrive from the new coordinator. If none arrives, it begins another election.

If a process $p_i$ receives a coordinator message, it sets its variable **elected$_i$** to the identifier of the coordinator contained within it and treats that process as the coordinator. If a process receives an election message, it sends back an **answer** message and begins another election – unless it has begun one already.

When a process is started to replace a crashed process, it begins an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes.

Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the '**bully**' algorithm.

When p1 ’s timeout period T’ expires (which we assume occurs before p2 ’s timeout expires), it deduces the absence of a **coordinator** message and begins another election. Eventually, p2 is elected coordinator (stage 4).




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

This algorithm clearly meets the liveness condition E2, by the assumption of reliable message delivery. And if no process is replaced, then the algorithm meets condition E1.

It is impossible for two processes to decide that they are the coordinator, since the process with the lower identifier will discover that the other exists and defer to it.

The algorithm is not guaranteed to meet the safety condition E1 if processes that have crashed are replaced by processes with the same identifiers. A process that replaces a crashed process p may decide that it has the highest identifier just as another process (which has detected p's crash) decides that it has the highest identifier. Two processes will therefore announce themselves as the coordinator concurrently.

Unfortunately, there are no guarantees on message delivery order, and the recipients of these messages may reach different conclusions on which is the coordinator process. Furthermore, condition E1 may be broken if the assumed timeout values turn out to be inaccurate – that is, if the processes' failure detector is unreliable.

In the best case the process with the second-highest identifier notices the coordinator's failure. Then it can immediately elect itself and send N – 2 coordinator messages. The turnaround time is one message.

The bully algorithm requires O ($N^2$) messages in the worst case – that is, when the process with the lowest identifier first detects the coordinator's failure. For then N – 1 processes altogether begin elections, each sending messages to processes with higher identifiers.

# Transactions

- In some situations, clients require a sequence of separate requests to a server to be atomic in the sense that:
  - 1. They are free from interference by operations being performed on behalf of other concurrent clients.
  - 2. Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

In below example, the first two actions transfer $100 from *A* to *B* and the second two transfer $200 from *C* to *B*. A client achieves a transfer operation by doing a withdrawal followed by a deposit.

A client's banking transaction

```
Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);
```

- Transactions originate from database management systems, where a transaction is an execution of a program that accesses a database.

- Transactions were introduced to distributed systems in the form of transactional file servers.

- Transactions on distributed objects consists of the execution of a sequence of client requests. From the client's point of view, a transaction is a sequence of operations that forms a single step, transforming the server data from one consistent state to another.

- Transactions can be provided as a part of middleware allowing clients' transactions to include multiple objects at multiple servers.

- The client is provided with operations to specify the beginning and end of a transaction. The client maintains a context for each transaction, which it propagates with each operation in that transaction.

- In all the above context, transaction applies to recoverable objects and is intended to be atomic, often called an **atomic transaction**. There are two aspects to atomicity
  - **i. All or nothing:** A transaction either completes successfully, in which case the effects of all of its operations are recorded in the objects, or has no effect at all. This all-or-nothing effect has two further aspects of its own:
    - **Failure atomicity:** The effects are atomic even when the server crashes.
    - **Durability:** After a transaction has completed successfully, all its effects are saved in permanent storage.

  - **ii. Isolation:** Each transaction must be performed without interference from other transactions; in other words, the intermediate effects of a transaction must not be visible to other transactions.

- To support failure atomicity and durability, the objects must be **recoverable.** When a server process crashes, the changes of completed transactions must be available in permanent storage so that when the server is replaced by a new process, it can recover the objects to reflect the all-or-nothing effect.

- A server that supports transactions must synchronize the operations sufficiently to ensure that the isolation requirement is met. One way of doing this is to perform the transactions serially – one at a time, in some arbitrary order.

- The aim for any server that supports transactions is to maximize concurrency. Therefore transactions are allowed to execute concurrently if this would have the same effect as a serial execution – that is, if they are **serially equivalent or serializable**.

- **ACID** properties of Transactions:
  - **A**tomicity: a transaction must be all or nothing;
  - **C**onsistency: a transaction takes the system from one consistent state to another consistent state;
  - **I**solation;
  - **D**urability.

  **Consistency** of transactions is generally the responsibility of the programmers of servers and clients to ensure that transactions leave the database consistent.

- Transaction capabilities can be added to servers of recoverable objects. Each transaction is created and managed by a coordinator, which implements the **Coordinator** interface.

Operations in the *Coordinator* interface

*openTransaction( ) → trans;*
Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

*closeTransaction(trans)→ (commit, abort);*
Ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

*abortTransaction(trans);*
Aborts the transaction.

- A transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator. The client specifies the sequence of invocations on recoverable objects that are to comprise a transaction.

- To achieve this, the client sends with each invocation the transaction identifier returned by **openTransaction**.

- When transactions are provided as middleware, the TID can be passed implicitly with all remote invocations between **openTransaction** and **closeTransaction** or **abortTransaction**.

- If the transaction has progressed normally, the reply states that the transaction is **committed** – this constitutes that all of the changes requested in the transaction are permanently recorded and that any future transactions that access the same data will see the results of all of the changes made during the transaction.

- When a transaction is aborted the parties involved (the recoverable objects and the coordinator) must ensure that none of its effects are visible to future transactions, either in the objects or in their copies in permanent storage.

**Transaction life histories**

| *Successful* | *Aborted by client* | *Aborted by server* | |
|---|---|---|---|
| *openTransaction* | *openTransaction* | | *openTransaction* |
| *operation* | *operation* | | *operation* |
| *operation* | *operation* | | *operation* |
| • | • | server aborts | • |
| • | • | *transaction* → | • |
| *operation* | *operation* | | *operation ERROR reported to client* |
| *closeTransaction* | *abortTransaction* | | |

**Service actions related to process crashes**

- If a server process crashes unexpectedly, it is eventually replaced. The new server process aborts any uncommitted transactions and uses a recovery procedure to restore the values of the objects to the values produced by the most recently committed transaction.

- To deal with a client that crashes unexpectedly during a transaction, servers can give each transaction an expiry time and abort any transaction that has not completed before its expiry time.

## Client actions related to server process crashes

- A server crashes while a transaction is in progress, the client will become aware of this when one of the operations returns an exception after a timeout.

- If a server crashes and is then replaced during the progress of a transaction, the transaction will no longer be valid and the client must be informed via an exception to the next operation.

## Concurrency control

Let us assume each of the operations **deposit, withdraw, getBalance** and **setBalance** is a synchronized operation – that is, that its effects on the instance variable that records the balance of an account are **atomic**.

Concurrent operations face the following problems:

**i. The lost update problem**

Consider the bank accounts *A*, *B* and *C*, whose initial balances are $100, $200 and $300, respectively. Transaction *T* transfers an amount from account *A* to account *B*. Transaction *U* transfers an amount from account *C* to

The lost update problem

| Transaction *T*: | | Transaction *U*: | |
|---|---|---|---|
| *balance = b.getBalance();* | | *balance = b.getBalance();* | |
| *b.setBalance(balance*1.1);* | | *b.setBalance(balance*1.1);* | |
| *a.withdraw(balance/10)* | | *c.withdraw(balance/10)* | |
| *balance = b.getBalance();* | $200 | | |
| | | *balance = b.getBalance();* | $200 |
| | | *b.setBalance(balance*1.1);* | $220 |
| *b.setBalance(balance*1.1);* | $220 | | |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/10)* | $280 |

account *B*. The amount transferred is calculated to increase the balance of *B* by 10%.

Now consider the effects of allowing the transactions *T* and *U* to run **concurrently**. This is an illustration of the 'lost update' problem. *U*'s update is lost because *T* overwrites it without seeing it. Both transactions have read the old value before either writes the new value.

## ii. Inconsistent retrievals

•Transaction *V* transfers a sum from account *A* to *B* and transaction *W* invokes the ***branchTotal*** method to obtain the sum of the balances of all the accounts in the bank.

The inconsistent retrievals problem

| Transaction *V*: | | Transaction *W*: | |
|---|---|---|---|
| *a.withdraw(100)*<br>*b.deposit(100)* | | *aBranch.branchTotal()* | |
| *a.withdraw(100);* | $100 | | |
| | | *total = a.getBalance()* | $100 |
| | | *total = total + b.getBalance()* | $300 |
| | | *total = total + c.getBalance()* | |
| *b.deposit(100)* | $300 | • | |
| | | • | |

•The balances of the two bank accounts, *A* and *B*, are both initially $200. The result of ***branchTotal*** includes the sum of *A* and *B* as $300, which is wrong. This is an illustration of the 'inconsistent retrievals' problem.

•*W*'s retrievals are inconsistent because *V* has performed only the withdrawal part of a transfer at the time the sum is calculated.

### iii. Serial equivalence

•An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a ***serially equivalent*** interleaving.

•A serially equivalent interleaving of two transactions produces the same effect as a serial one, hence the lost update problem can be solved by means of serial equivalence.

•The below fig shows one such interleaving in which the operations that affect the shared account, *B*, are actually serial, for transaction *T* does all its operations on *B* before transaction *U* does.

A serially equivalent interleaving of *T* and *U*

| Transaction *T*: | | Transaction *U*: | |
|---|---|---|---|
| *balance = b.getBalance()* | | *balance = b.getBalance()* | |
| *b.setBalance(balance*1.1)* | | *b.setBalance(balance*1.1)* | |
| *a.withdraw(balance/10)* | | *c.withdraw(balance/10)* | |
| *balance = b.getBalance()* | $200 | | |
| *b.setBalance(balance*1.1)* | $220 | | |
| | | *balance = b.getBalance()* | $220 |
| | | *b.setBalance(balance*1.1)* | $242 |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/10)* | $278 |

- A serially equivalent interleaving of a retrieval transaction and an update transaction, for example as in Figure, will prevent inconsistent retrievals occurring.

A serially equivalent interleaving of *V* and *W*

| **Transaction *V*:** | | **Transaction *W*:** | |
|---|---|---|---|
| *a.withdraw(100);*<br>*b.deposit(100)* | | *aBranch.branchTotal( )* | |
| *a.withdraw(100);* | $100 | | |
| *b.deposit(100)* | $300 | | |
| | | *total = a.getBalance( )* | $100 |
| | | *total = total + b.getBalance( )* | $400 |
| | | *total = total + c.getBalance( )* | |
| | | ... | |

## Conflicting operations

•A pair of operations *conflict* when their combined effect depends on the order in which they are executed.

•Basically, ***read*** accesses the value of an object and ***write*** changes its value The *effect* of an operation refers to the value of an object set by a *write* operation and the result returned by a *read* operation.

*Read* and *write* operation conflict rules

| *Operations of different transactions* | | *Conflict* | *Reason* |
|---|---|---|---|
| *read* | *read* | No | Because the effect of a pair of *read* operations does not depend on the order in which they are executed |
| *read* | *write* | Yes | Because the effect of a *read* and a *write* operation depends on the order of their execution |
| *write* | *write* | Yes | Because the effect of a pair of *write* operations depends on the order of their execution |

•For any pair of transactions, it is possible to determine the order of pairs of conflicting operations on objects accessed by both of them.

## Recoverability from aborts

•Servers must record all the effects of committed transactions and none of the effects of aborted transactions.

•Two problems associated with aborting transactions

•These problems are called **'dirty reads' and 'premature writes'**, and both of them can occur in the presence of serially equivalent executions of transactions.

## Dirty reads

A dirty read when transaction *T* aborts

| Transaction *T*: | | Transaction *U*: | |
|---|---|---|---|
| *a.getBalance()* | | *a.getBalance()* | |
| *a.setBalance(balance + 10)* | | *a.setBalance(balance + 20)* | |
| *balance = a.getBalance()* | $100 | | |
| *a.setBalance(balance + 10)* | $110 | | |
| | | *balance = a.getBalance()* | $110 |
| | | *a.setBalance(balance + 20)* | $130 |
| | | *commit transaction* | |
| *abort transaction* | | | |

- The ‘dirty read’ problem is caused by the interaction between a *read* operation in one transaction and an earlier *write* operation in another transaction on the same object.

- The two executions shown in fig are serially equivalent. Now suppose that the transaction *T* aborts after *U* has committed. Then the transaction *U* will have seen a value that never existed, since *A* will be restored to its original value.

- We say that the transaction *U* has performed a **dirty read**. As it has committed, it cannot be undone.

**Recoverability of transactions**

- If a transaction (like *U*) has committed after it has seen the effects of a transaction that subsequently aborted, the situation is not recoverable.

- The strategy for recoverability is to delay commits until after the commitment of any other transaction whose uncommitted state has been observed.

**Cascading aborts**

•Suppose that transaction *U* delays committing until after *T* aborts. Then, *U* must abort as well.

•All other transactions that have seen the effects due to *U*, they too must be aborted. The aborting of these latter transactions may cause still further transactions to be aborted. Such situations are called **cascading aborts**.

•To avoid cascading aborts, transactions are only allowed to read objects that were written by committed transactions.

**Premature writes**

•This one is related to the interaction between *write* operations on the same object belonging to different transactions. We consider two *setBalance* transactions, *T* and *U*, on account *A*.

•Before the transactions, the balance of account A was $100. The two executions are serially equivalent, with *T* setting the balance to $105 and *U* setting it to $110.

## Overwriting uncommitted values

| Transaction *T*: | | Transaction *U*: | |
|---|---|---|---|
| *a.setBalance(105)* | | *a.setBalance(110)* | |
| | $100 | | |
| *a.setBalance(105)* | $105 | | |
| | | *a.setBalance(110)* | $110 |

- If the transaction *U* aborts and *T* commits, the balance should be $105. Some database systems implement the action of *abort* by restoring **'before images'** of all the **writes** of a transaction.

- In our example, *A* is $100 initially, which is the 'before image' of *T*'s *write*; similarly, $105 is the 'before image' of *U*'s *write*. Thus if *U* aborts, we get the correct balance of $105.

- when *U* commits and then *T* aborts. The balance should be $110, but as the 'before image' of *T*'s *write* is $100, we get the wrong balance of $100.

- Similarly, if *T* aborts and then *U* aborts, the 'before image' of *U*'s *write* is $105 and we get the wrong balance of $105 – the balance should revert to $100.

- To ensure correct results in a recovery scheme that uses before images, write operations **must be delayed** until earlier transactions that updated the same objects have either **committed or aborted**.

**Strict executions of transactions**

- It is required that transactions delay both their *read* and *write* operations so as to avoid both dirty reads and premature writes.

- The executions of transactions are called *strict* if the service delays both *read* and *write* operations on an object until **all transactions that previously wrote that object have either committed or aborted**.

## Tentative versions

• For a server with recoverable objects to participate in transactions, it must be designed so that any updates of objects can be removed if and when a transaction aborts.

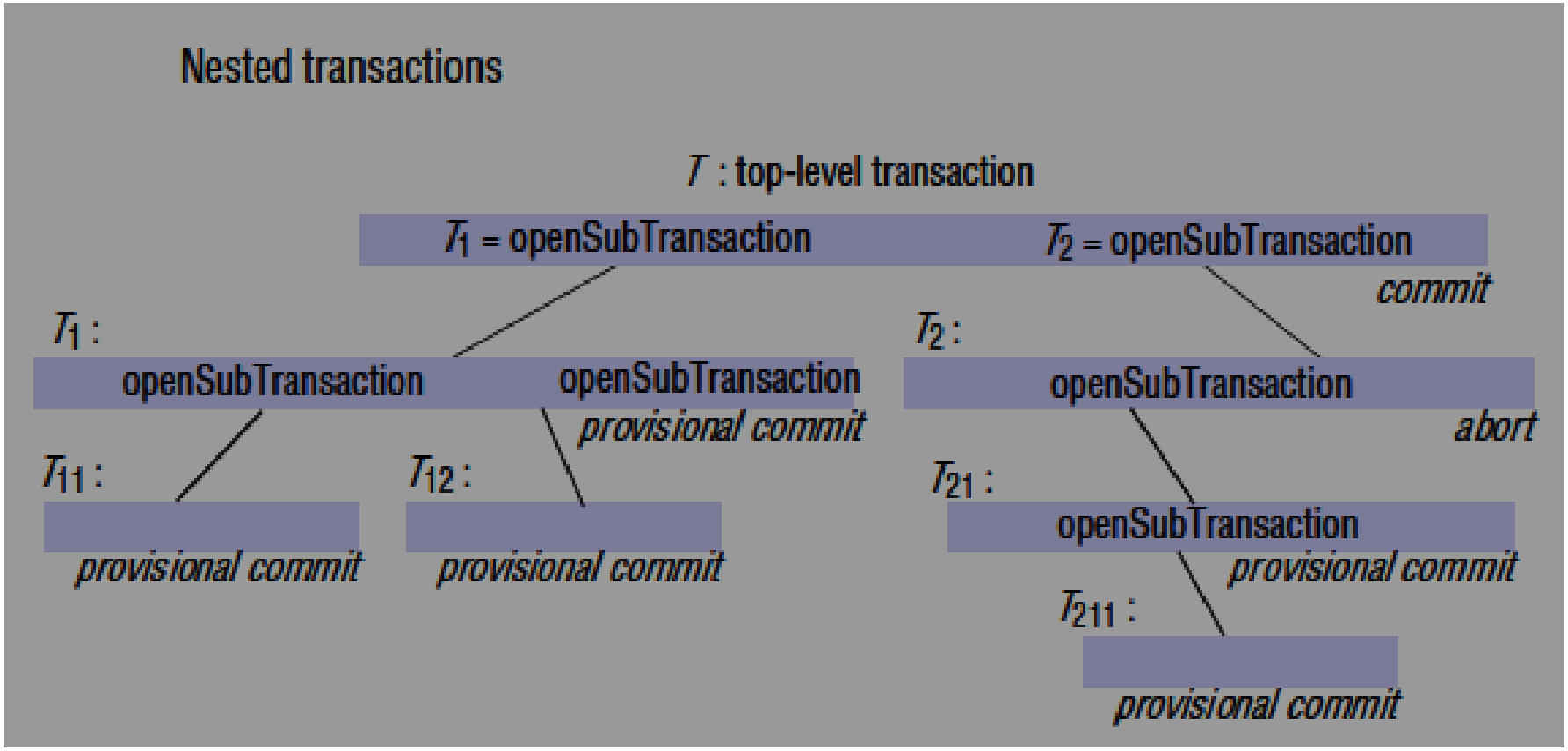
•To make this possible, all of the update operations performed during a transaction are done in **tentative versions of objects in volatile memory**.

•The tentative versions are transferred to the objects only when a transaction commits, by which time they will also have been recorded in permanent storage.

## Nested transactions

•Nested transactions extend the above transaction model by allowing transactions to be composed of other transactions. The outermost transaction in a set of nested transactions is called the **top-level** transaction.

•Transactions other than the top-level transaction are called **subtransactions**.




- Subtransactions at the same level, such as *T1* and *T2*, can run concurrently, but their access to common objects is serialized.

- Each subtransaction can fail independently of its parent and of the other subtransactions.

- When a subtransaction aborts, the parent transaction can sometimes choose an alternative subtransaction to complete its task.

- A normal transaction is referred as **flat transaction**. It is flat because all of its work is done at the same level between an openTransaction and a commit or abort, and it is not possible to commit or abort parts of it.

**Advantages of Nested transactions:**

- Subtransactions at one level (and their descendants) may run concurrently with other subtransactions at the same level in the hierarchy. When subtransactions run in different servers, they can work in parallel.

- Subtransactions can commit or abort independently. a set of nested subtransactions is potentially more robust. In fact, a parent can decide on different actions according to whether a subtransaction has aborted or not.

**Rules for committing of nested transactions:**

1.A transaction may commit or abort only after its child transactions have completed.

2. When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. Its decision to abort is final.

3.When a parent aborts, all of its subtransactions are aborted.

4. When a subtransaction aborts, the parent can decide whether to abort or not. If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted

## Locks

•Transactions must be scheduled so that their effect on shared data is serially equivalent. A server can achieve serial equivalence of transactions by serializing access to the objects.

•A simple example of a serializing mechanism is the use of exclusive locks. In this locking scheme, the server attempts to lock any object that is about to be used by any operation of a client's transaction.

•If a client requests access to an object that is already locked due to another client's transaction, the request is suspended and the client must wait until the object is unlocked.

Transactions *T* and *U* with exclusive locks

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance = b.getBalance()* | | *balance = b.getBalance()* | |
| *b.setBalance(bal*1.1)* | | *b.setBalance(bal*1.1)* | |
| *a.withdraw(bal/10)* | | *c.withdraw(bal/10)* | |
| Operations | Locks | Operations | Locks |
| *openTransaction* | | | |
| *bal = b.getBalance()* | lock *B* | | |
| *b.setBalance(bal*1.1)* | | *openTransaction* | |
| *a.withdraw(bal/10)* | lock *A* | *bal = b.getBalance()* | waits for *T*'s lock on *B* |
| *closeTransaction* | unlock *A*, *B* | • • • | |
| | | | lock *B* |
| | | *b.setBalance(bal*1.1)* | |
| | | *c.withdraw(bal/10)* | lock *C* |
| | | *closeTransaction* | unlock *B*, *C* |

- All pairs of conflicting operations of two transactions should be executed in the same order. To ensure this, a transaction is not allowed any new locks after it has released a lock.

- The first phase of each transaction is a ‘growing phase’, during which new locks are acquired. In the second phase, the locks are released (a ‘shrinking phase’). This is called **two-phase locking**.

- Under a **strict execution** regime, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted.

- To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. This is called **strict two-phase locking.**

- Two types of locks are used: Before a transaction’s *read* operation is performed, a **read lock** should be set on the object. Before a transaction’s *write* operation is performed, a **write lock** should be set on the object.

- Whenever it is impossible to set a lock immediately, the transaction (and the client) must wait until it is possible to do so – a client's request is never rejected. Read locks are sometimes called **shared locks**.

The **operation conflict rules** states that:

1. If a transaction *T* has already performed a *read* operation on a particular object, then a concurrent transaction *U* must not *write* that object until *T* commits or aborts.

2. If a transaction *T* has already performed a *write* operation on a particular object, then a concurrent transaction *U* must not *read* or *write* that object until *T* commits or aborts.

- To enforce condition 1, a request for a write lock on an object is delayed by the presence of a read lock belonging to another transaction. To enforce condition 2, a request for either a read lock or a write lock on an object is delayed by the presence of a write lock belonging to another transaction.

Lock compatibility

| *For one object* | | *Lock requested* | |
|---|---|---|---|
| | | *read* | *write* |
| *Lock already set* | *none* | OK | OK |
| | *read* | OK | wait |
| | *write* | wait | wait |

•The entry in each cell shows the effect on a transaction that requests the type of lock given above when the object has been locked in another transaction with the type of lock on the left.

•Inconsistent retrievals and lost updates are caused by conflicts between *read* operations in one transaction and *write* operations in another without the protection of a concurrency control scheme such as locking.

•Inconsistent retrievals are prevented by performing the retrieval transaction before or after the update transaction

The **rules for the use of locks** in a strict two-phase locking implementation are summarized:

## Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:

   (a) If the object is not already locked, it is locked and the operation proceeds.

   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.

   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.

   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule b is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

- Lock promotion refers to the conversion of a lock to a stronger lock – that is, a lock that is more exclusive. The read lock allows other read locks, whereas the write lock does not.

- Neither allows other write locks. Therefore, a write lock is more exclusive than a read lock. Locks may be promoted because the result is a more exclusive lock.

- Locking is performed when the requests for *read* and *write* operations are about to be applied to the recoverable objects, and unlocking is performed by the *commit* or *abort* operations of the transaction coordinator.

**Lock implementation**

- The granting of locks will be implemented by a separate object in the server, called the **lock manager**. The lock manager holds a set of locks, for example in a hash table.

- Each lock is an instance of the class *Lock* and is associated with a particular object.

Lock class

```
public class Lock {
    private Object object;          // the object being protected by the lock
    private Vector holders;         // the TIDs of current holders
    private LockType lockType;  // the current type

    public synchronized void acquire(TransID trans, LockType aLockType ){
        while(/*another transaction holds the lock in conflicting mode*/) {
            try {
                wait();
            }catch ( InterruptedException e){/*...*/ }
        }
        if (holders.isEmpty()) { // no TIDs hold lock
            holders.addElement(trans);
            lockType  = aLockType;
        } else if (/*another transaction holds the lock, share it*/ ) ){
            if (/* this transaction not a holder*/) holders.addElement(trans);
        } else if (/* this transaction is a holder but needs a more exclusive lock*/)
                lockType.promote();
        }
    }

    public synchronized void release(TransID trans ){
        holders.removeElement(trans);     // remove this holder
        // set locktype to none
        notifyAll();
    }
}
```

- Each instance of *Lock* maintains the following information in its instance variables:
  - The identifier of the locked object;
  - The transaction identifiers of the transactions that currently hold the lock (shared locks can have several holders);
  - A lock type.

- The methods of *Lock* are synchronized so that the threads attempting to acquire or release a lock will not interfere with one another. But, in addition, attempts to acquire the lock use the *wait* method whenever they have to wait for another thread to release it.

- *acquire* method arguments specify a transaction identifier and the type of lock required by that transaction.

- It tests whether the request can be granted. If another transaction holds the lock in a conflicting mode, it invokes *wait*, which causes the caller's thread to be suspended until a corresponding *notify*.

- When, eventually, the condition is satisfied, the remainder of **the method sets the lock appropriately:**
  - if no other transaction holds the lock, just add the given transaction to the holders and set the type;
  - else if another transaction holds the lock, share it by adding the given transaction to the holders (unless it is already a holder);
  - else if this transaction is a holder but is requesting a more exclusive lock, promote the lock.

- The *release* method's arguments specify the transaction identifier of the transaction that is releasing the lock. It removes the transaction identifier from the holders, sets the lock type to *none* and calls *notifyAll*.

- The method notifies all waiting threads in case there are multiple transactions waiting to acquire read locks – all of them may be able to proceed.
- The class **LockManager** is shown in Fig:

## *LockManager* class

```
public class LockManager {
   private Hashtable theLocks;

   public  void setLock(Object object, TransID trans,  LockType lockType){
      Lock foundLock;
      synchronized(this){
            // find the lock associated with object
            // if there isn't one, create it and add it to the hashtable
         }
      foundLock.acquire(trans, lockType);
   }

  // synchronize this one because we want to remove all entries
   public synchronized void unLock(TransID trans) {
      Enumeration e = theLocks.elements();
      while(e.hasMoreElements()){
          Lock aLock = (Lock)(e.nextElement());
          if(/* trans is a holder of this lock*/ ) aLock.release(trans);
       }
   }
}
```

- All requests to set locks and to release them on behalf of transactions are sent to an instance of **LockManager**:
  - The **setLock** method's arguments specify the object that the given transaction wants to lock and the type of lock. It finds a lock for that object in its hash table or, if necessary, creates one. It then invokes the *acquire* method of that lock.

  - The **unLock** method's argument specifies the transaction that is releasing its locks. It finds all of the locks in the hash table that have the given transaction as a holder. For each one, it calls the *release* method.

**Locking rules for nested transactions**

- The aim of a locking scheme for nested transactions is to serialize access to objects.

1. Each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions.

2. Each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set

- The first rule is enforced by arranging that every lock that is acquired by a successful subtransaction is *inherited* by its parent when it completes. Inherited locks are also inherited by ancestors.

- The top-level transaction eventually inherits all of the locks that were acquired by successful subtransactions at any depth in a nested transaction.

- This ensures that the locks can be held until the top-level transaction has committed or aborted, which prevents members of different sets of nested transactions observing one another's partial effects.

- The **second rule** is enforced as follows:
- Parent transactions are not allowed to run concurrently with their child transactions. If a parent transaction has a lock on an object, it *retains* the lock during the time that its child transaction is executing. This means that the child transaction temporarily acquires the lock from its parent for its duration.

2. Subtransactions at the same level are allowed to run concurrently, so when they access the same objects, the locking scheme must serialize their access.

**Lock acquisition and release rules:**

- For a subtransaction to acquire a read lock on an object, no other active transaction can have a write lock on that object, and the only retainers of a write lock are its ancestors.

- For a subtransaction to acquire a write lock on an object, no other active transaction can have a read or write lock on that object, and the only retainers of read and write locks on that object are its ancestors.

- When a subtransaction commits, its locks are inherited by its parent, allowing the parent to retain the locks in the same mode as the child.

- When a subtransaction aborts, its locks are discarded. If the parent already retains the locks, it can continue to do so.

  Subtransactions at the same level that access the same object will take turns to acquire the locks retained by their parent. This ensures that their access to a common object is serialized.

## Deadlocks

•The use of locks can lead to deadlock.

Deadlock with write locks

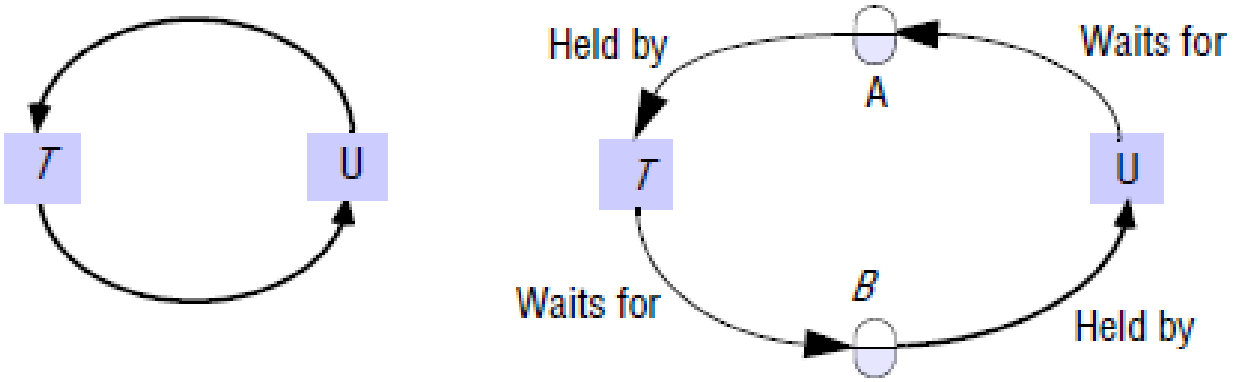
| Transaction *T* | | Transaction *U* | |
|---|---|---|---|
| Operations | Locks | Operations | Locks |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| ••• | waits for *U*'s lock on *B* | *a.withdraw(200);* | waits for *T*'s lock on *A* |
| ••• | | ••• | |
| ••• | | ••• | |

## Definition of deadlock

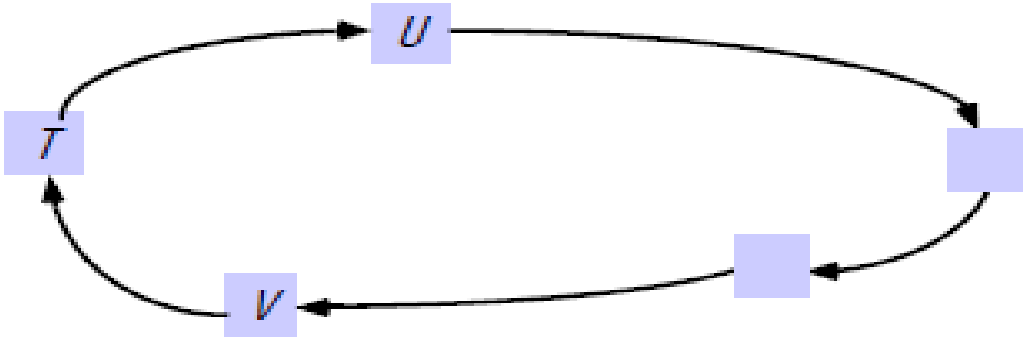
•Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock. A *wait-for graph* can be used to represent the waiting relationships between current transactions.

- In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions. – there is an edge from node *T* to node *U* when transaction *T* is waiting for transaction *U* to release a lock.

The wait-for graph for Figure 16.19




A cycle in a wait-for graph



- A wait-for graph contains a cycle $T \rightarrow U \rightarrow ... \rightarrow V \rightarrow T$ . Each transaction is waiting for the next transaction in the cycle. All of these transactions are blocked waiting for locks. None of the locks can ever be released, and the transactions are deadlocked.

- If one of the transactions in a cycle is aborted, then its locks are released and that cycle is broken.

**Deadlock prevention**

- One solution is to lock all of the objects used by a transaction when it starts. This would need to be done as a single atomic step so as to avoid deadlock at this stage. This approach unnecessarily restricts access to shared resources.

- Deadlocks can also be prevented by requesting locks on objects in a predefined order, but this can result in premature locking and a reduction in concurrency.

## Upgrade locks

- CORBA's Concurrency Control Service introduces a third type of lock, called *upgrade*, the use of which is intended to avoid deadlocks.

- Deadlocks are often caused by two conflicting transactions first taking read locks and then attempting to promote them to write locks.

- A transaction with an upgrade lock on a data item is permitted to read that data item, but this lock conflicts with any upgrade locks set by other transactions on the same data item.

- This type of lock cannot be set implicitly by the use of a *read* operation, but must be requested by the client.

## Deadlock detection

- Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.

- When a deadlock is detected, one of the transactions in the cycle must be chosen and then be aborted. The corresponding node and the edges involving it must be removed from the wait-for graph. This will happen when the aborted transaction has its locks removed.

- The choice of the transaction to abort is not simple. Some factors that may be taken into account are the age of the transaction and the number of cycles in which it is involved.

**Timeouts**

- Lock timeouts are a method for resolution of deadlocks that is commonly used. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable.

- If any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken (that is, the object is unlocked) and the waiting transaction resumes. The transaction whose lock has been broken is normally aborted.

- There are many problems with the use of timeouts as a remedy for deadlocks: the worst problem is that transactions are sometimes aborted due to their locks becoming vulnerable when other transactions are waiting for them, but there is actually no deadlock.

- In an overloaded system, the number of transactions timing out will increase, and transactions taking a long time can be penalized. In addition, it is hard to decide on an appropriate length for a timeout.

**Increasing concurrency in locking schemes**

- Even when locking rules are based on the conflicts between *read* and *write* operations and the granularity at which they are applied is as small as possible, there is still some scope for increasing concurrency. There are 2 approaches:

**i. Two-version locking**

- This is an optimistic scheme that allows one transaction to write tentative versions of objects while other transactions read from the committed versions of the same objects.

- *read* operations only wait if another transaction is currently committing the same object.

- Transactions cannot commit their *write* operations immediately if other uncompleted transactions have read the same objects.

- Therefore, transactions that request to commit in such a situation are made to wait until the reading transactions have completed.

- Deadlocks may occur when transactions are waiting to commit. Therefore, transactions may need to be aborted when they are waiting to commit, to resolve deadlocks.

- This variation on strict two-phase locking uses **three types of lock**: a read lock, a write lock and a commit lock.

- Before a transaction's *read* operation is performed, a read lock must be set on the object – the attempt to set a read lock is successful unless the object has a commit lock, in which case the transaction waits.

- Before a transaction's *write* operation is performed, a write lock must be set on the object – the attempt to set a write lock is successful unless the object has a write lock or a commit lock, in which case the transaction waits.

**Hierarchic locks**

- In some applications, the granularity suitable for one operation is not appropriate for another operation.

- Gray proposed the use of a hierarchy of locks with different granularities. At each level, the setting of a parent lock has the same effect as setting all the equivalent child locks.

- In Gray's scheme, each node in the hierarchy can be locked, giving the owner of the lock explicit access to the node and giving implicit access to its children.

- Before a child node is granted a read-write lock, an intention to read-write lock is set on the parent node and its ancestors (if any). The intention lock is compatible with other intention locks but conflicts with read and write locks

Lock compatibility table for hierarchic locks

| *For one object* | | *Lock to be set* | | | |
|---|---|---|---|---|---|
| | | *read* | *write* | *I-read* | *I-write* |
| *Lock already set* | *none* | OK | OK | OK | OK |
| | *read* | OK | wait | OK | wait |
| | *write* | wait | wait | wait | wait |
| | *I-read* | OK | wait | OK | OK |
| | *I-write* | wait | wait | OK | OK |

- Gray also proposed a third type of intention lock – one that combines the properties of a read lock with an intention to write lock.

- Hierarchic locks have the advantage of reducing the number of locks when mixed granularity locking is required. The compatibility tables and the rules for promoting locks are more complex.

- The mixed granularity of locks could allow each transaction to lock a portion whose size is chosen according to its needs. A long transaction that accesses many objects could lock the whole collection, whereas a short transaction can lock at finer granularity.

## Optimistic concurrency control

**Drawbacks of locking:**

- Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Even read-only transactions (queries), which cannot possibly affect the integrity of the data, must, in general, use locking in order to guarantee that the data being read is not modified by other transactions at the same time. But locking may be necessary only in the worst case.

- The use of locks can result in deadlock. Deadlock prevention reduces concurrency severely, and therefore deadlock situations must be resolved either by the use of timeouts or by deadlock detection. Neither of these is wholly satisfactory for use in interactive programs.

- To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce significantly the potential for concurrency.

Alternative approach proposed by Kung and Robinson:

- The likelihood of two clients' transactions accessing the same object is low. Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a *closeTransaction* request.

- When a conflict arises, some transaction is generally aborted and will need to be restarted by the client.

- Each transaction has the following **phases**:

i. **Working phase**: During the working phase, each transaction has a tentative version of each of the objects that it updates. This is a copy of the most recently committed version of the object.

- The use of tentative versions allows the transaction to abort (with no effect on the objects), either during the working phase or if it fails validation due to other conflicting transactions.

- *read* operations are performed immediately – if a tentative version for that transaction already exists, a *read* operation accesses it; otherwise, it accesses the most recently committed value of the object.

- *write* operations record the new values of the objects as tentative values (which are invisible to other transactions). When there are several concurrent transactions, several different tentative values of the same object may coexist.

- In addition, two records are kept of the objects accessed within a transaction: a *read set* containing the objects read by the transaction and a *write set* containing the objects written by the transaction. Note that as all *read* operations are performed on committed versions of the objects (or copies of them), dirty reads cannot occur.

**ii. Validation phase:** When the *closeTransaction* request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects.

- If the validation is successful, then the transaction can commit. If the validation fails, then some form of conflict resolution must be used and either the current transaction or, in some cases, those with which it conflicts will need to be aborted.

iii. **Update phase:** If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions of the objects have been recorded in permanent storage.

**Validation of transactions**

- Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other *overlapping* transactions – that is, any transactions that had not yet committed at the time this transaction started.

- To assist in performing validation, each transaction is assigned a transaction number when it enters the validation phase (that is, when the client issues a *closeTransaction*).

- If the transaction is validated and completes successfully, it retains this number; if it fails the validation checks and is aborted, or if the transaction is read only, the number is released for reassignment.

| $T_v$ | $T_i$ | *Rule* | |
|---|---|---|---|
| *write* | *read* | 1. | $T_i$ must not read objects written by $T_v$. |
| *read* | *write* | 2. | $T_v$ must not read objects written by $T_i$. |
| *write* | *write* | 3. | $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$. |

- A transaction *Tv* to be serializable with respect to an overlapping transaction *Ti*, their operations must conform to the **following rules**:

- To prevent overlapping, the entire validation and update phases can be implemented as a critical section so that only one client at a time can execute it.

- In order to increase concurrency, part of the validation and updating may be implemented outside the critical section, but it is essential that the assignment of transaction numbers is performed sequentially.

- There are two forms of validation
  - **Backward validation** checks the transaction undergoing validation with other preceding overlapping transactions – those that entered the validation phase before it.

  - **Forward validation** checks the transaction undergoing validation with other later transactions, which are still active.

**Backward validation**

- As all the *read* operations of earlier overlapping transactions were performed before the validation of *Tv* started, they cannot be affected by the *writes* of the current transaction (and rule 1 is satisfied).

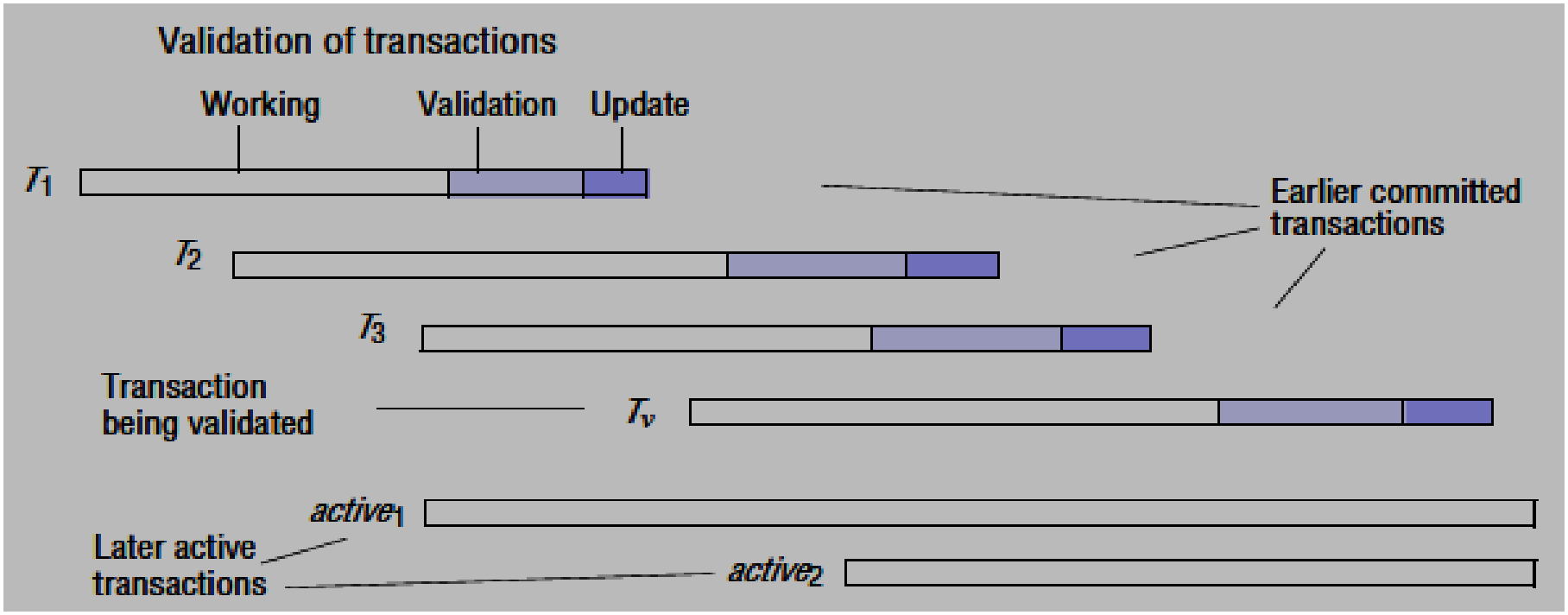
- The validation of transaction *Tv* checks whether its read set (the objects affected by the *read* operations of *Tv*) overlaps with any of the write sets of earlier overlapping transactions, *Ti* (rule 2).

- If there is any overlap, the validation fails.

- Let *startTn* be the biggest transaction number assigned (to some other committed transaction) at the time when transaction *Tv* started its working phase and *finishTn* be the biggest transaction number assigned at the time when *Tv* entered the validation phase.

- The following program describes the algorithm for the validation of *Tv*:

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
    if (read set of Tv intersects write set of Ti) valid = false;
}
```



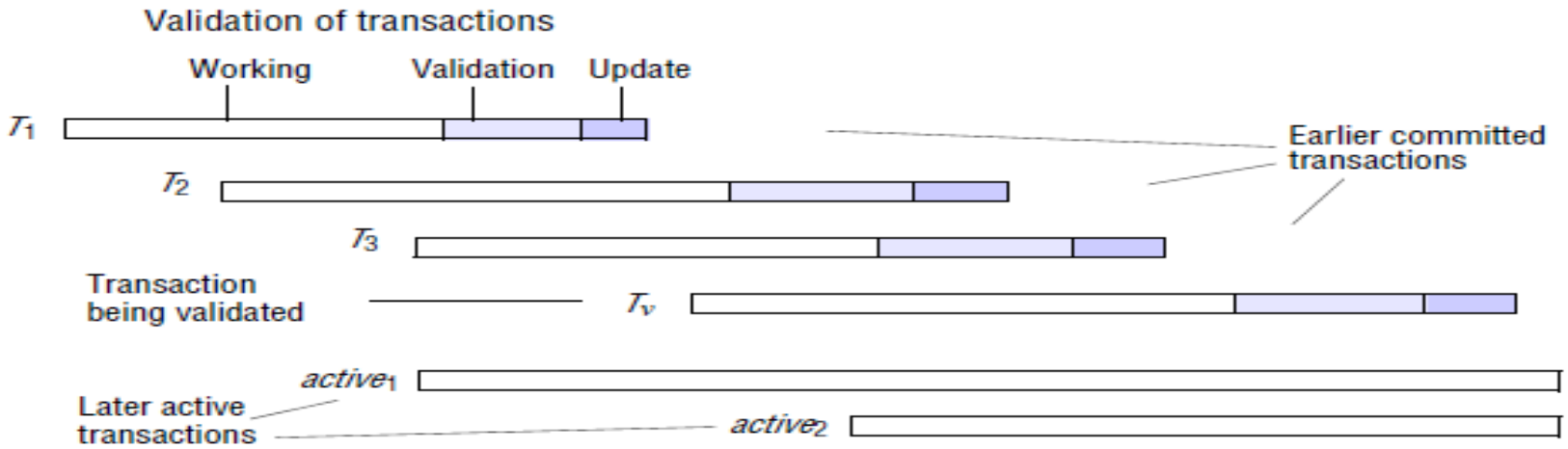
- Overlapping transactions that might be considered in the validation of a transaction *T*v. Time increases from left to right. The earlier committed transactions are *T*1, *T*2 and *T*3. *T*1 committed before *Tv* started. *T*2 and *T*3 committed before *Tv* finished its working phase. *StartTn* + 1 = *T*2 and *finishTn* = *T*3. In backward validation, the read set of *Tv* must be compared with the write sets of *T*2 and *T*3.

- In backward validation, the read set of the transaction being validated is compared with the write sets of other transactions that have already committed. Therefore, the only way to resolve any conflicts is to abort the transaction that is undergoing validation.

- In backward validation, transactions that have no *read* operations (only *write* operations) need not be checked.

- **Forward validation**
- In forward validation of the transaction *Tv*, the write set of *Tv* is compared with the read sets of all overlapping active transactions – those that are still in their working phase (rule 1).
- Rule 2 is automatically fulfilled because the active transactions do not write until after *Tv* has completed. Let the active transactions have (consecutive) transaction identifiers *active*1 to *activeN*.
- The following program describes the algorithm for the forward validation of *Tv*:

```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
      if (write set of Tv intersects read set of Tid) valid = false;
}
```



- The write set of transaction *Tv* must be compared with the read sets of the transactions with identifiers *active*1 and *active*2.
- Forward validation should allow for the fact that read sets of active transactions may change during validation and writing.
- **Comparison of forward and backward validation**
- Forward validation allows flexibility in the resolution of conflicts
- Backward validation allows only one choice – to abort the transaction being validated.
- In general, the read sets of transactions are much larger than the write sets. Therefore, backward validation
- Compares a possibly large read set against the old write sets
- Whereas forward validation checks a small write set against the read sets of active transactions.

- **Starvation**
- When a transaction is aborted, it will normally be restarted by the client program.
- But in schemes that rely on aborting and restarting transactions, there is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of objects each time it is restarted.
- The prevention of a transaction ever being able to commit is called starvation.
- Kung and Robinson suggest that this could be done if the server detects a transaction that has been aborted several times. They suggest that when the server detects such a transaction it should be given exclusive access by the use of a critical section protected by a semaphore.

## Timestamp ordering

In concurrency control schemes based on timestamp ordering, each operation in a transaction is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately and can then be restarted by the client.

Each transaction is assigned a unique timestamp value when it starts. The timestamp defines its position in the time sequence of transactions. Requests from transactions can be totally ordered according to their timestamps.

The basic timestamp ordering rule is based on operation conflicts,

> A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.

This rule assumes that there is only one version of each object and restricts access to one transaction at a time. If each transaction has its own tentative version of each object it accesses, then multiple concurrent transactions can access the same object.

The timestamp ordering rule is refined to ensure that each transaction accesses a consistent set of versions of the objects.

the *write* operations are recorded in tentative versions of objects and are invisible to other transactions until a *closeTransaction* request is issued and the transaction is committed.

Every object has a write timestamp and a set of tentative versions each of which has a write timestamp associated with it; each object also has a set of read timestamps.

The write timestamp of the (committed) object is earlier than that of any of its tentative versions, and the set of read timestamps can be represented by its maximum member.

Whenever a transaction's *write* operation on an object is accepted, the server creates a new tentative version of the object with its write timestamp set to the transaction timestamp.

A transaction's *read* operation is directed to the version with the maximum write timestamp less than the transaction timestamp. Whenever a transaction's *read* operation on an object is accepted, the timestamp of the transaction is added to its set of read timestamps.

When a transaction is committed, the values of the tentative versions become the values of the objects, and the timestamps of the tentative versions become the timestamps of the corresponding objects.

In timestamp ordering, each request by a transaction for a *read* or *write* operation on an object is checked to see whether it conforms to the operation conflict rules.

A request by the current transaction *Tc* can conflict with previous operations done by other transactions, *Ti*, whose timestamps indicate that they should be later than *Tc*.

Operation conflicts for timestamp ordering

| *Rule* | $T_c$ | $T_i$ | |
|---|---|---|---|
| 1. | *write* | *read* | $T_c$ must not *write* an object that has been *read* by any $T_i$ where $T_i > T_c$. This requires that $T_c \geq$ the maximum read timestamp of the object. |
| 2. | *write* | *write* | $T_c$ must not *write* an object that has been *written* by any $T_i$ where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object. |
| 3. | *read* | *write* | $T_c$ must not *read* an object that has been *written* by any $T_i$ where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object. |

## Timestamp ordering write rule

By combining rules 1 and 2 we get the following rule for deciding whether to accept a *write* operation requested by transaction *Tc* on object *D*.

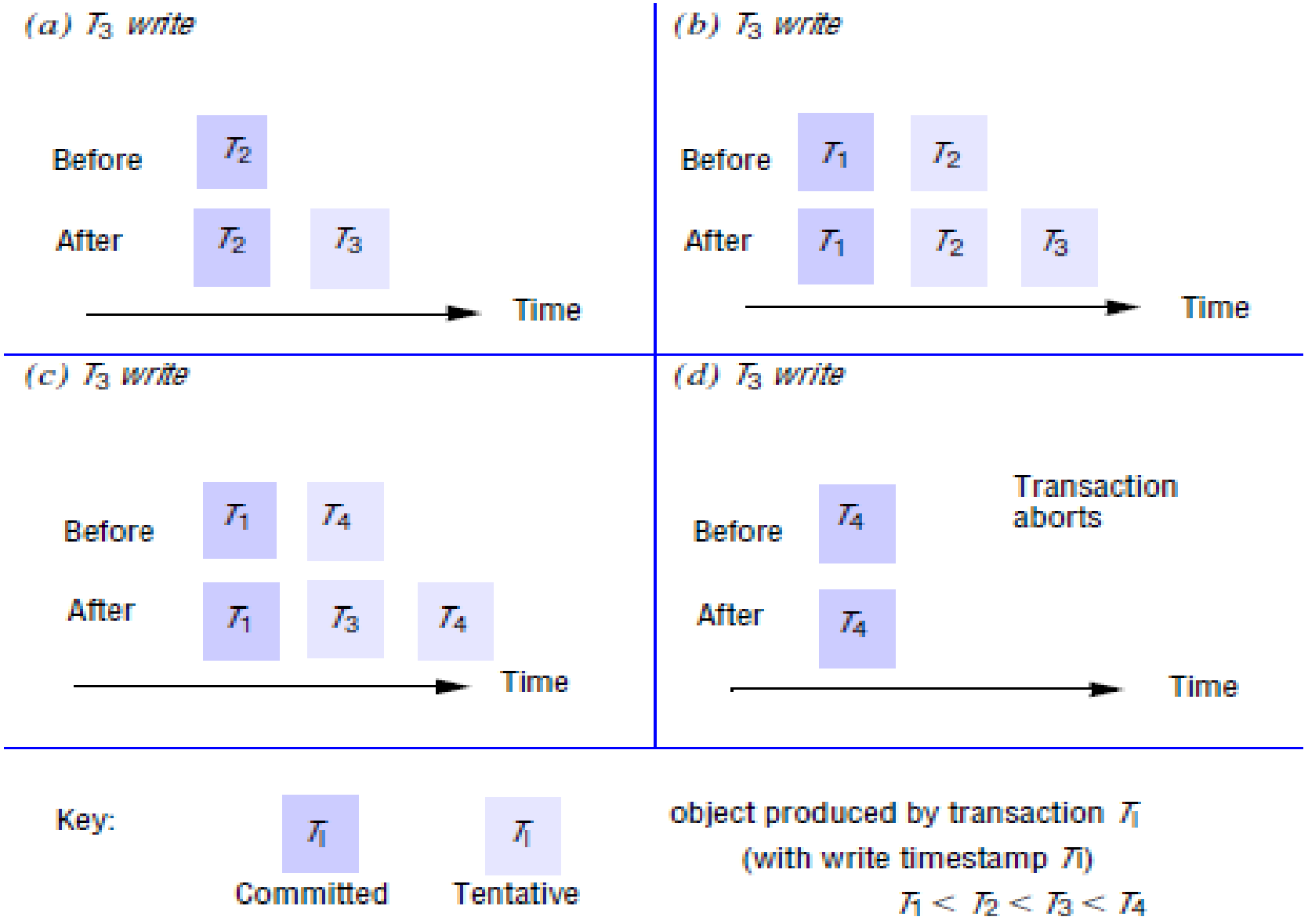
if ($T_c \geq$ maximum read timestamp on $D$ &&
    $T_c >$ write timestamp on committed version of $D$)
        perform *write* operation on tentative version of $D$ with write timestamp $T_c$
else /* write is too late */
    Abort transaction $T_c$

If a tentative version with write timestamp *Tc* already exists, the *write* operation is addressed to it; otherwise, a new tentative version is created and given write timestamp *Tc*.

In cases (a) to (c), *T*3 > write timestamp on the committed version of the object and a tentative version with write timestamp *T*3 is inserted at the appropriate place in the list of tentative versions ordered by their transaction timestamps.

In case (d), *T*3 < write timestamp on the committed version of the object and the transaction is aborted.

## Write operations and timestamps

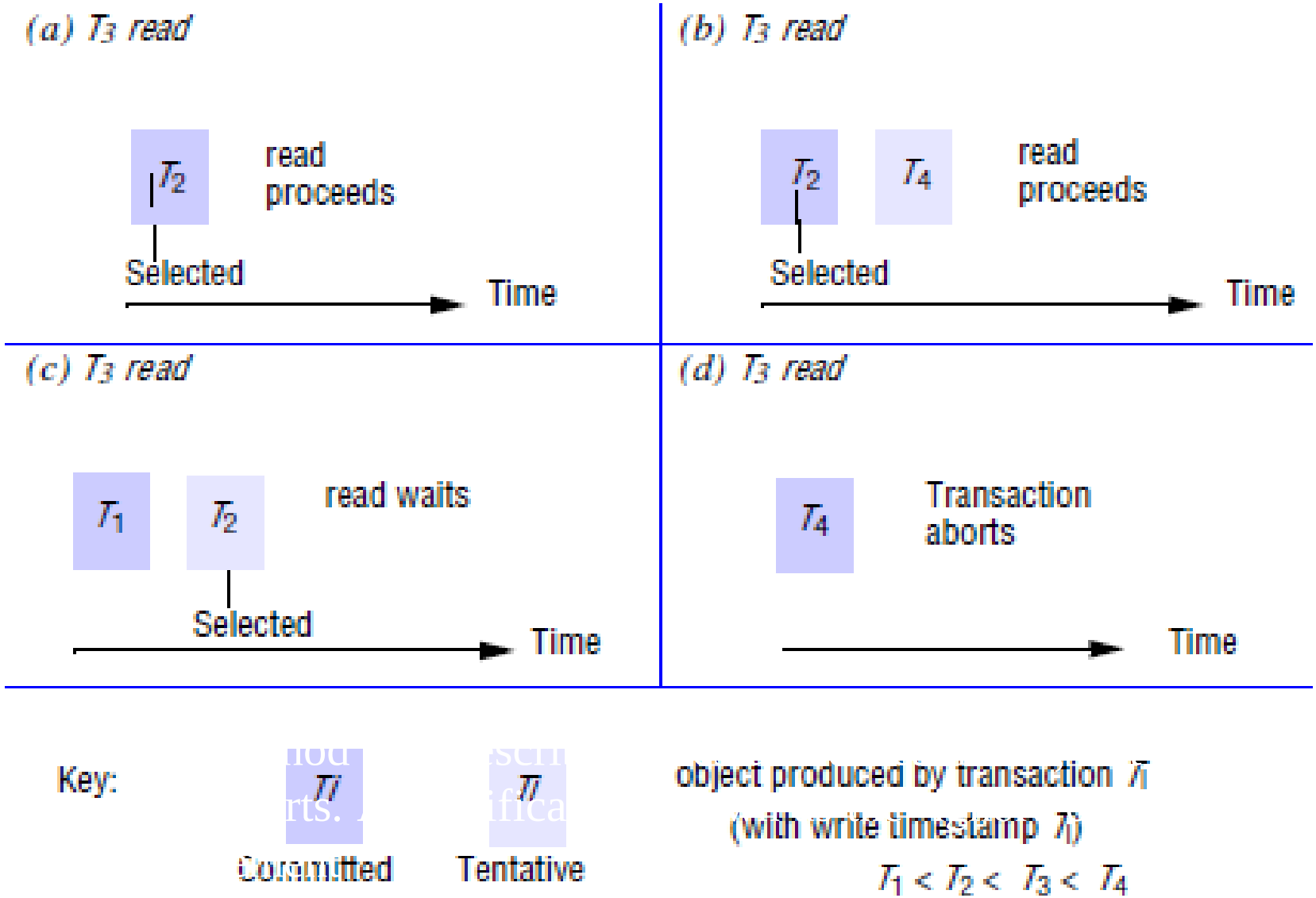


## Timestamp ordering read rule

By using rule 3 we arrive at the following rule for deciding whether to accept immediately, to wait or to reject a *read* operation requested by transaction *Tc* on object D.

```
if ( Tc > write timestamp on committed version of D) {
    let Dselected be the version of D with the maximum write timestamp δ Tc
    if (Dselected is committed)
        perform read operation on the version Dselected
    else
        wait until the transaction that made version Dselected commits or aborts
        then reapply the read rule
} else
    Abort transaction Tc
```

*Read* operations and timestamps



## Atomic commit protocols

- The atomicity property of transactions requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them.
- In the case of a distributed transaction, the client has requested operations at more than one server.
- A transaction comes to an end when the client requests that transaction be committed or aborted.

One phase atomic commit protocol

The coordinator communicates the commit or abort request to all of the participants in the transaction and it keep on repeating the request until all of them have acknowledged that they have carried it out .

Drawback:

It does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit

**Two-phase commit protocol** :

In the first phase of the protocol,

each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a prepared state for a transaction if it will eventually be able to commit it.

In the second phase of the protocol,

every participant in the transaction carries out the joint decision. If any one participant votes to abort, then the decision must be to abort the transaction. If all the participants vote to commit, then the decision is to commit the transaction

## Failure model for the commit protocols

Crash failures of processes are masked by **replacing a crashed process with a new process** whose state is set from information saved in permanent storage and information held by other processes

## The two-phase commit protocol:

When the client asks the coordinator to commit the transaction the two-phase commit protocol comes into use.

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; in the second, it tells them to commit (or abort) the transaction

If a participant can commit its part of a transaction, it will agree as soon as it has recorded the changes it has made (to the objects) and its status in permanent storage

## Operations for two-phase commit protocol

*canCommit?(trans)→ Yes / No*

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

*doCommit(trans)*

Call from coordinator to participant to tell participant to commit its part of a transaction.

*doAbort(trans)*

Call from coordinator to participant to tell participant to abort its part of a transaction.

*haveCommitted(trans, participant)*

Call from participant to coordinator to confirm that it has committed the transaction.

*getDecision(trans) → Yes / No*

Call from participant to coordinator to ask for the decision on a transaction when it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

The methods **canCommit**, **doCommit** and **doAbort** are methods in the interface of the participant.

The methods **haveCommitted** and **getDecision** are in the coordinator interface

## The two-phase commit protocol

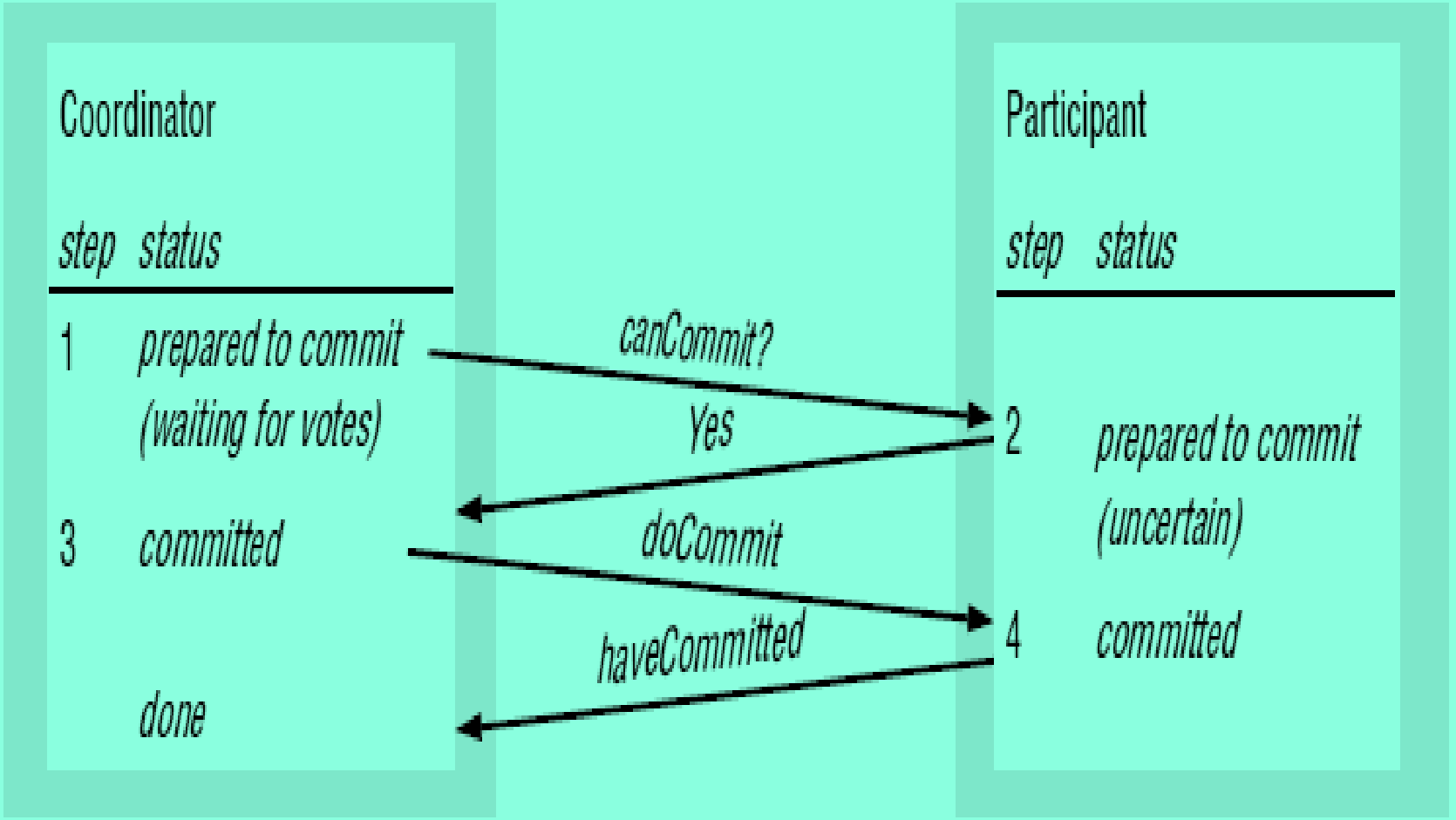
*Phase 1 (voting phase):*

1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No*, the participant aborts immediately.

*Phase 2 (completion according to outcome of vote):*

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes*, the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise, the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

**Figure 17.6** Communication in two-phase commit protocol



## Timeout actions in the two-phase commit protocol

Consider first the situation where a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort the Transaction  Such a participant is *uncertain* of the outcome and cannot proceed any further until it gets the outcome of the vote from the coordinator.

The participant cannot decide unilaterally what to do next, and meanwhile the objects used by its transaction cannot be released for use by other transactions. The participant can make a *getDecision* request to the coordinator to determine the outcome of the transaction. When it gets the reply, it continues the protocol at step 4

If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the uncertain state

Alternative strategy:

The participants obtain a decision cooperatively instead of contacting the coordinator. These strategies have the advantage that they may be used when the coordinator has failed

Drawback:

With a cooperative protocol, if all the participants are in the *uncertain* state, they will be unable to get a decision until the coordinator or a participant with the necessary knowledge is available

**Performing Abort Operation:**

Case 1:

A participant may be delayed when it has carried out all its client requests in the transaction but has not yet received a *canCommit?* call from the coordinator.

As the client sends the *closeTransaction* to the coordinator, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time – for example, by the time a timeout period on a lock expires.

As no decision has been made at this stage, the participant can decide to *abort* unilaterally.

Case 2:

The coordinator may be delayed when it is waiting for votes from the participants

As it has not yet decided the fate of the transaction it may decide to abort the transaction after some period of time

It must then announce *doAbort* to the participants who have already sent their votes

## Two-phase commit protocol for nested transactions

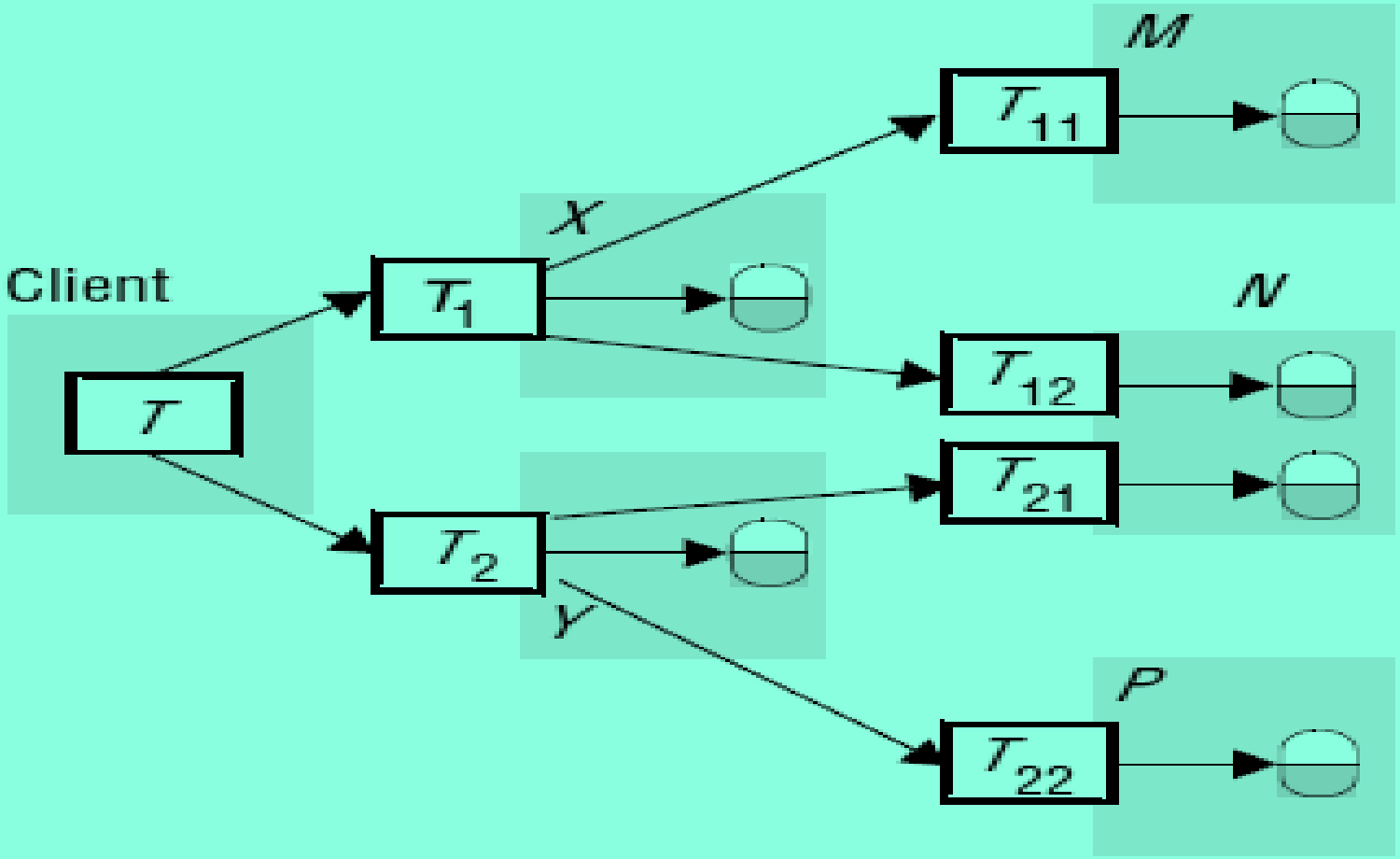
The outermost transaction in a set of nested transactions is called the top-level transaction.

Transactions other than the top-level transaction are called sub transactions.

T is the top-level transaction and T1, T2, T11, T12, T21 and T22 are subtransactions. T1 and T2 are child transactions of T, which is referred to as their parent. Similarly, T11 and T12 are child transactions of T1, and T21 and T22 are child transactions of T2. Each sub transaction starts after its parent and finishes before it. Thus, for example, T11 and T12 start after T1 and finish before it.

When a sub transaction completes, it makes an independent decision either to commit provisionally or to abort. A provisional commit is different from being prepared to commit: nothing is backed up in permanent storage. If the server crashes subsequently, its replacement will not be able to commit. After all sub transactions have completed, the provisionally committed ones participate in a two-phase commit protocol, in which servers of provisionally committed sub transactions express their intention to commit and those with an aborted ancestor will abort.

(b) Nested transactions



Being prepared to commit guarantees a sub transaction will be able to commit, whereas a provisional commit only means it has finished correctly – and will probably agree to commit when it is subsequently asked to.

## Operations in coordinator for nested transactions

*openSubTransaction(trans)* → *subTrans*

Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

*getStatus(trans)*→ *committed, aborted, provisional*

Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed*, *aborted* or *provisional*.

A client starts a set of nested transactions by opening a top-level transaction with an **openTransaction** operation, which returns a transaction identifier for the top-level transaction.

The client starts a subtransaction by invoking the **openSubTransaction** operation, whose argument specifies its parent transaction.

The new subtransaction automatically joins the parent transaction, and a transaction identifier for a subtransaction is returned.

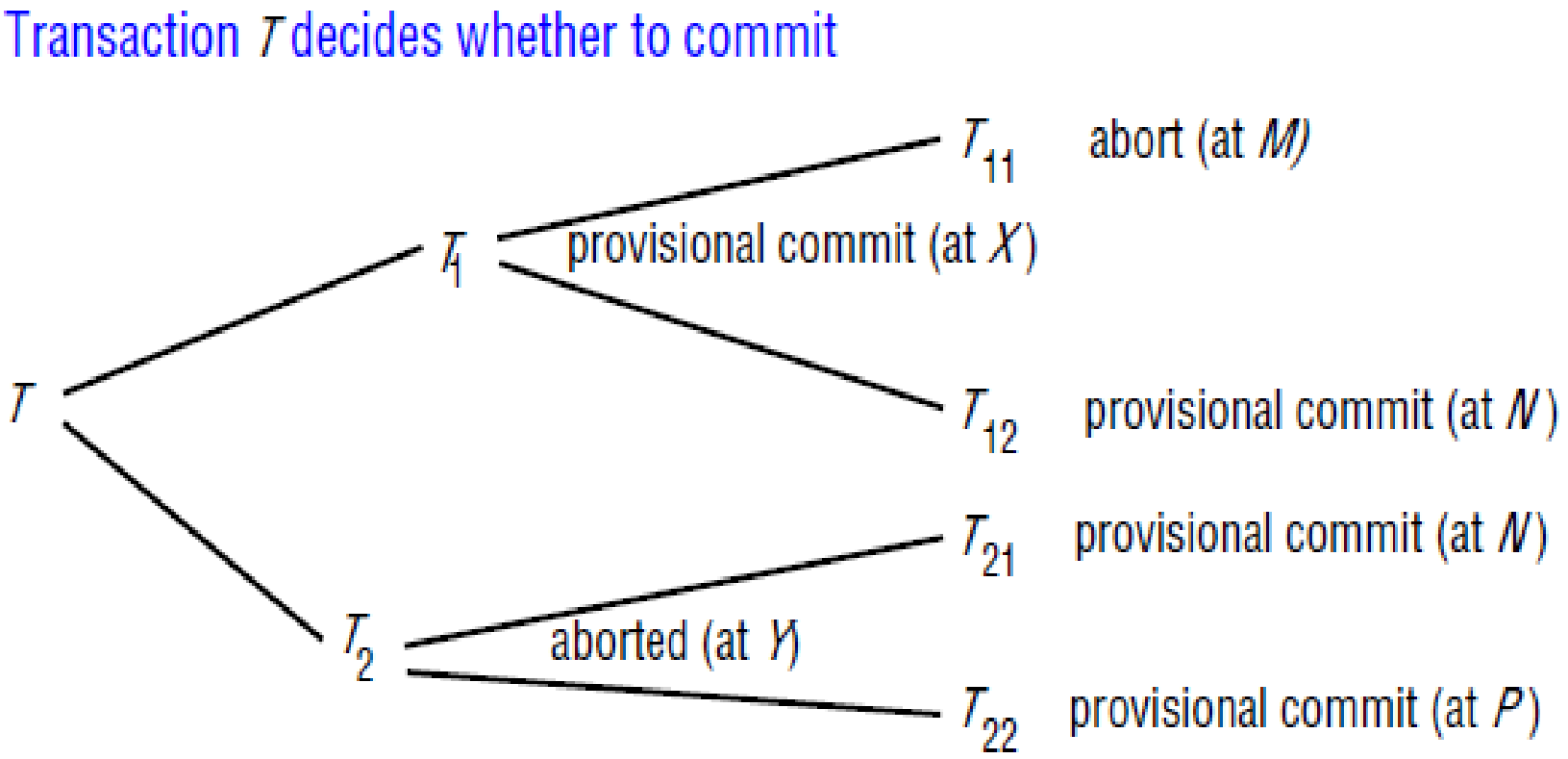
An identifier for a subtransaction must be an extension of its parent's TID, constructed in such a way that the identifier of the parent or top-level transaction of a subtransaction can be determined from its own transaction identifier.

All subtransaction identifiers should be globally unique.

The client makes a set of nested transactions come to completion by invoking **closeTransaction** or **abortTransaction** on the coordinator of the top-level transaction.

A parent transaction – including a top-level transaction – can commit even if one of its child subtransactions has aborted.

In such cases, the parent transaction will be programmed to take different actions according to whether a subtransaction has committed or aborted



# Information held by coordinators of nested transactions

| *Coordinator of transaction* | *Child transactions* | *Participant* | *Provisional commit list* | *Abort list* |
|---|---|---|---|---|
| $T$ | $T_1$, $T_2$ | yes | $T_1$, $T_{12}$ | $T_{11}$, $T_2$ |
| $T_1$ | $T_{11}$, $T_{12}$ | yes | $T_1$, $T_{12}$ | $T_{11}$ |
| $T_2$ | $T_{21}$, $T_{22}$ | no (aborted) | | $T_2$ |
| $T_{11}$ | | no (aborted) | | $T_{11}$ |
| $T_{12}$, $T_{21}$ | | $T_{12}$ but not $T_{21}$* | $T_{21}$, $T_{12}$ | |
| $T_{22}$ | | no (parent aborted) | $T_{22}$ | |

* $T_{21}$'s parent has aborted

Types of Two Phase Commit Protocols

1. **Hierarchic two-phase commit protocol**

   In this approach, the two-phase commit protocol becomes a multi-level nested protocol. The coordinator of the top-level transaction communicates with the coordinators of the subtransactions for which it is the immediate parent. It sends **canCommit?** messages to each of the latter, which in turn pass them on to the coordinators of their child transactions (and so on down the tree). Each participant collects the replies from its descendants before replying to its parent

2. Flat two-phase commit protocol

   In this approach, the coordinator of the top-level transaction sends **canCommit?** messages to the coordinators of all of the subtransactions in the provisional commit list

## Distributed deadlocks

Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks

Using **timeouts** to resolve possible deadlocks is a **clumsy** approach – it is difficult to choose an appropriate timeout interval, and transactions may be aborted unnecessarily.

With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock. Most deadlock detection schemes operate by finding cycles in the transaction wait-for graph

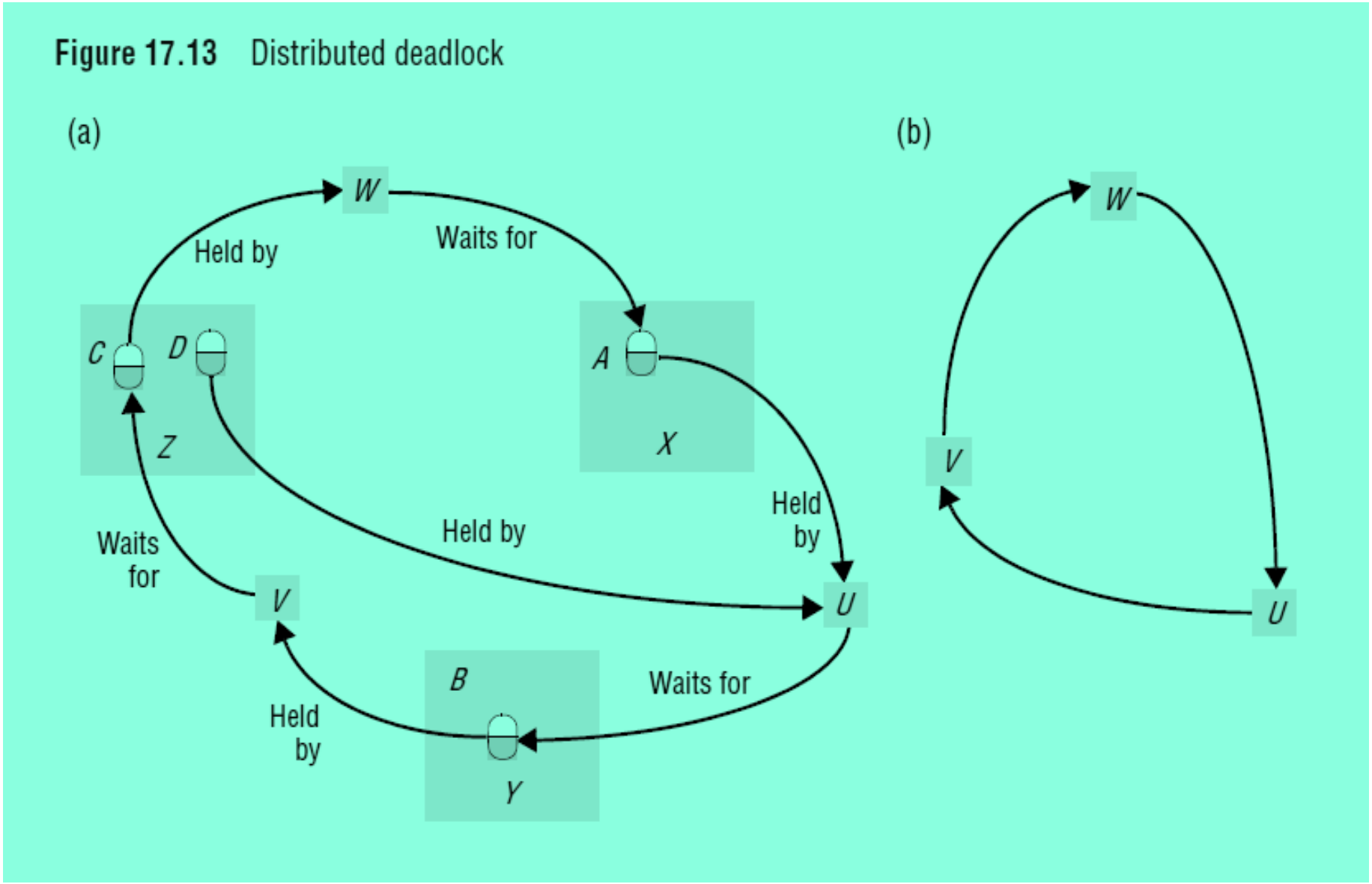
In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for graph can in theory be constructed from the local ones.

There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a *distributed deadlock*

**Figure 17.12** Interleavings of transactions *U*, *V* and *W*

| *U* | | *V* | | *W* | |
|---|---|---|---|---|---|
| *d.deposit(10)* | lock *D* | | | | |
| | | *b.deposit(10)* | lock *B* | | |
| *a.deposit(20)* | lock *A* | | at *Y* | | |
| | at *X* | | | | |
| | | | | *c.deposit(30)* | lock *C* |
| *b.withdraw(30)* | wait at *Y* | | | | at *Z* |
| | | *c.withdraw(20)* | wait at *Z* | | |
| | | | | *a.withdraw(20)* | wait at *X* |

# The complete wait-for graph



Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions.

Local wait-for graphs can be built by the lock manager at each server the local wait-for graphs of the servers are:
server *Y*: *U* →*V* (added when *U* requests *b.withdraw(30)*)
server *Z*: *V* → *W* (added when *V* requests *c.withdraw(20)*)
server *X*: *W* → *U* (added when *W* requests *a.withdraw(20)*)

As the global wait-for graph is held in part by each of the several servers involved, communication between these servers is required to find cycles in the graph.

A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector.

From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for graph

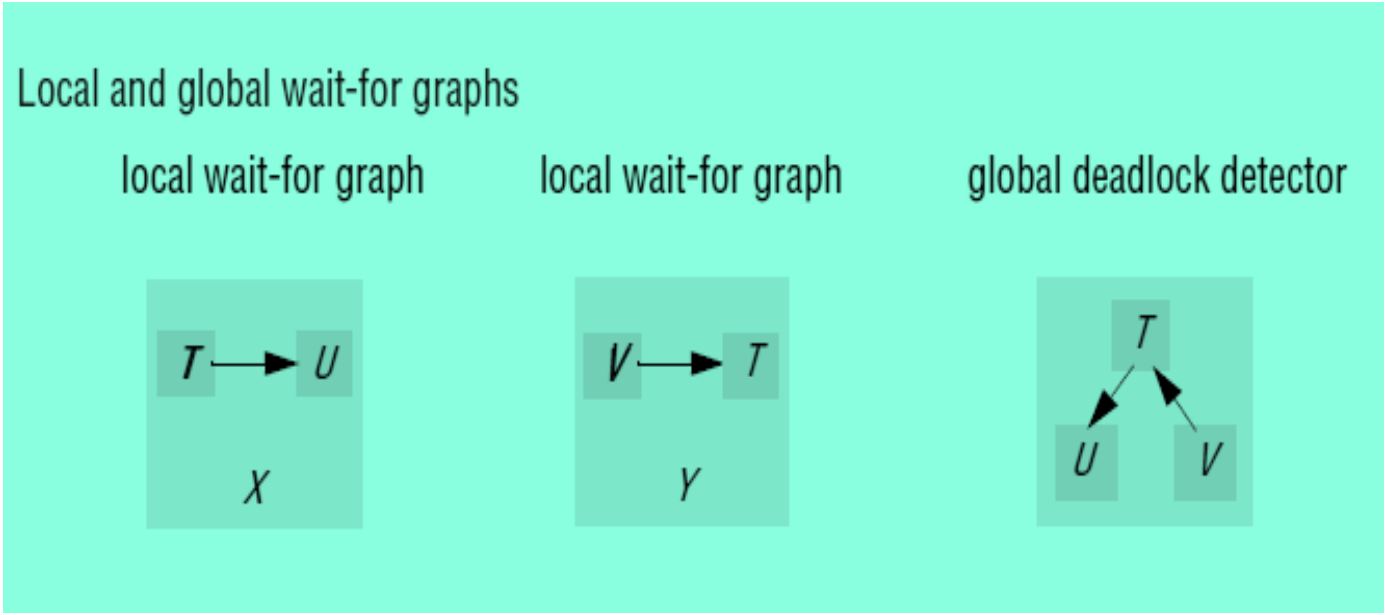
The global deadlock detector checks for cycles in the global wait-for graph. When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort.

Centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. It suffers from the usual problems associated with centralized solutions in distributed systems **poor availability, lack of fault tolerance and no ability to scale.**

In addition, the **cost** of the frequent transmission of local wait-for graphs is **high.** If the global graph is collected less frequently, deadlocks may take **longer to be detected.**

**Phantom deadlock :**

A deadlock that is ‘detected’ but is not really a deadlock is called a phantom deadlock




Consider the case of a global deadlock detector that receives local wait-for graphs from servers *X* and *Y*.

Suppose that transaction *U* then releases an object at server *X* and requests the one held by *V* at server *Y*.

Suppose also that the global detector receives server *Y*'s local graph before server *X*'s. In this case, it would detect a cycle $T \rightarrow U \rightarrow V \rightarrow T$, although the edge $T \rightarrow U$ no longer exists. This is an example of a phantom deadlock

**Edge chasing:**

A distributed approach to deadlock detection uses a technique called *edge chasing* or *path pushing*.

In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges.

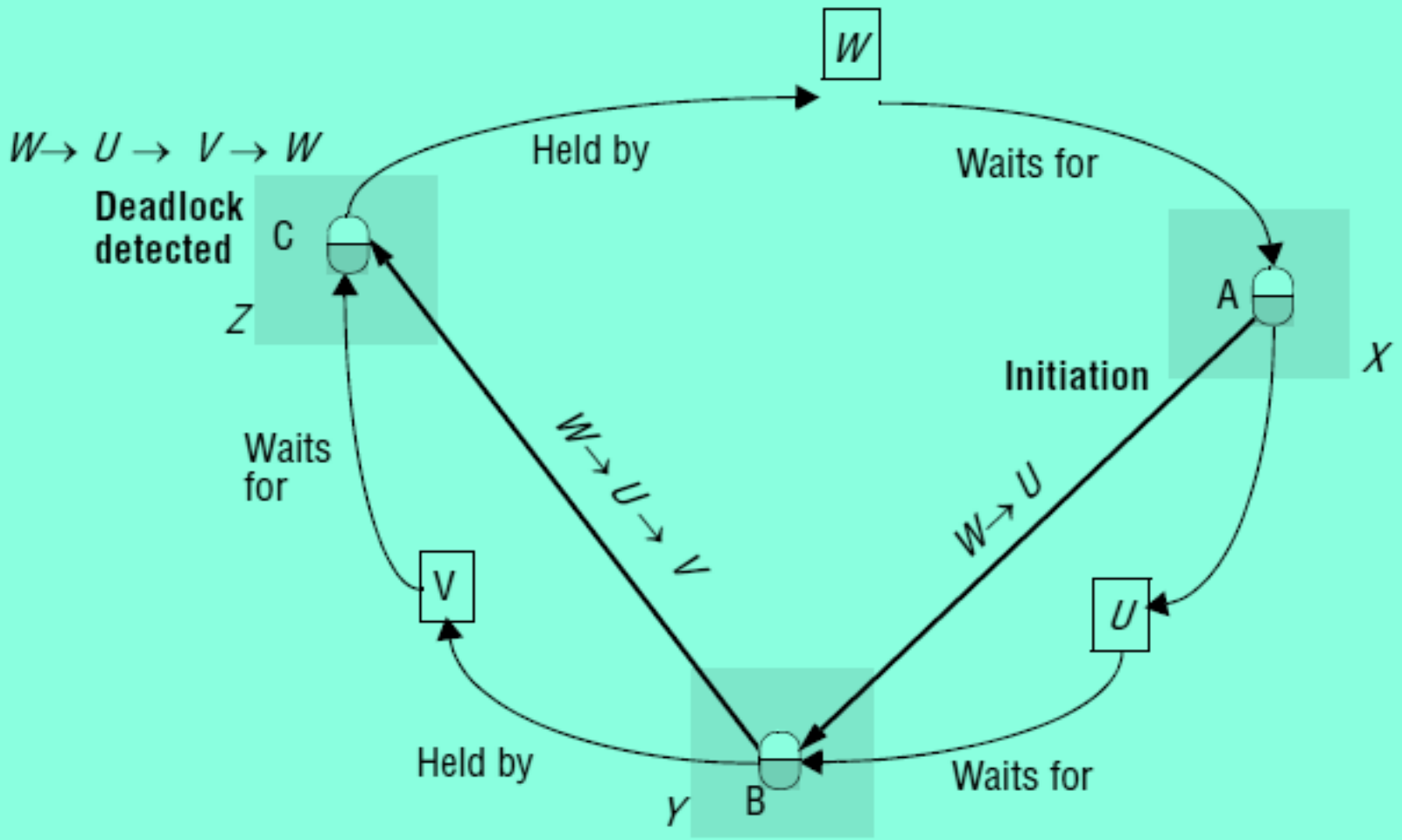
The servers attempt to find cycles by forwarding messages called *probes*, which follow the edges of the graph throughout the distributed system.

A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph.

Edge-chasing algorithms have three steps

1. Initiation
2. Detection
3. Resolution

Probes transmitted to detect deadlock



**Initiation**:

When a server notes that a transaction T starts waiting for another transaction U, where U is waiting to access an object at another server, it initiates detection by sending a probe containing the edge < T →U > to the server of the object at which transaction U is blocked.

If U is sharing a lock, probes are sent to all the holders of the lock. Sometimes further transactions may start sharing the lock later on, in which case probes can be sent to them too.

**Detection**:

Detection consists of receiving probes and deciding whether a deadlock has occurred and whether to forward the probes.

For example, when a server of an object receives a probe < T →U > (indicating that T is waiting for a transaction U that holds a local object), it checks to see whether U is also waiting.

If it is, the transaction it waits for (for example, V) is added to the probe (making it < T →U→ V > , and if the new transaction (V) is waiting for another object elsewhere, the probe is forwarded.

In this way, paths through the global wait-for graph are built one edge at a time.

Before forwarding a probe, the server checks to see whether the transaction (for example, T) it has just added has caused the probe to contain a cycle (for example, < T →U →V →T >).

If this is the case, it has found a cycle in the graph and a deadlock has been detected.

**Resolution:**

When a cycle is detected, a transaction in the cycle is aborted to break the deadlock

**Example:**

The following steps describe how deadlock detection is initiated and the probes that are forwarded during the corresponding detection phase:

Server X initiates detection by sending probe < W →U > to the server of B (Server Y).

Server Y receives probe < W →U >, notes that B is held by V and appends V to the probe to produce < W →U →V >. It notes that V is waiting for C at server Z. This probe is forwarded to server Z.

Server Z receives probe < W →U → V >, notes C is held by W and appends W to the probe to produce < W → U → V → W >.

This path contains a cycle. The server detects a deadlock. One of the transactions in the cycle must be aborted to break the deadlock. The transaction to be aborted can be chosen according to transaction priorities

**Transaction priorities:**

In order to ensure that only one transaction in a cycle is aborted, transactions are given *priorities* in such a way that all transactions are totally ordered.
Ex. Timestamps

When a deadlock cycle is found, the transaction with the lowest priority is aborted. Even if several different servers detect the same cycle, they will all reach the same decision as to which transaction is to be aborted

It might appear that transaction priorities could also be used to reduce the number of situations that cause deadlock detection to be initiated, by using the rule that detection is initiated only when a higher-priority transaction starts to wait for a lower-priority one.

Transaction priorities could also be used to reduce the number of probes that are forwarded. The general idea is that probes should travel ‘downhill’ – that is, from transactions with higher priorities to transactions with lower priorities

## Replication

It is the maintenance of copies of data at multiple computers. Ex. the caching of resources from web servers in browsers and web proxy servers.

## Motivations for replication

1. **Performance enhancement** – Performance is enhanced thru the caching of data at clients and servers to avoid the latency of fetching resources from the originating server. Data are sometimes replicated transparently between several originating servers in the same domain. The workload is shared between the servers by binding all the server IP addresses to the site's DNS name

   Replication of immutable data is trivial: it increases performance with little cost to the system. Replication of changing data, such as that of the Web, incurs overheads in the form of protocols designed to ensure that clients receive up-to-date data.

   Thus there are limits to the effectiveness of replication as a performance-enhancement technique.

2. **Increased availability**

Users require services to be highly available. That is, the proportion of time for which a service is accessible with reasonable response times should be close to 100%. The factors that are relevant to high availability are:

**i. server failures:**

If data are replicated at two or more failure-independent servers, then client software may be able to access data at an alternative server should the default server fail or become unreachable. That is, the percentage of time during which the *service* is available can be enhanced by replicating server data.

An important difference between caching systems and server replication is that caches do not necessarily hold collections of objects such as files in their entirety. So caching does not necessarily enhance availability at the application level.

**ii. Network partitions and Disconnected operation**

Network partitions and disconnected operation are the second factor that militate(prevent) against high availability. Mobile users may deliberately

disconnect their computers or become unintentionally disconnected from a wireless network as they move around.

For example, a user on a train with a laptop may have no access to networking (wireless networking may be interrupted, or they may have no such capability). In order to be able to work in these circumstances – so-called **disconnected working** or **disconnected operation** – the user will often prepare by copying heavily used data, such as the contents of a **shared diary**, from their usual environment to the laptop

**Trade-off** to availability during such a period of disconnection:
when the user consults or updates the diary, they risk reading data that someone else has altered in the meantime. Disconnected working is only feasible if the user can cope with **stale** data and can later resolve any conflicts that arise.

**3. Fault tolerance**

Highly available data is not necessarily strictly correct data. It may be out of date, for example; or two users on opposite sides of a network partition may make updates that conflict and need to be resolved. A fault-tolerant service,

by contrast, always guarantees strictly correct behaviour despite a certain number and type of faults.

The correctness concerns the freshness of data supplied to the client, the effects of the client's operations upon the data the timeliness of the service's responses. The same basic technique used for high availability – that of replicating data and functionality between computers – is also applicable for achieving fault tolerance.

The system must manage the coordination of its components precisely to maintain the correctness guarantees in the face of failures, which may occur at any time.

**Requirements for Replicated Data:**

i. **Replication transparency -** clients should **not** normally have to be aware that multiple physical copies of data exist. As far as clients are concerned, data are organized as individual logical objects and they identify only one item in each case when they request an operation to be performed.

Furthermore, clients expect operations to return only one set of values, despite the fact that operations may be performed upon more than one physical copy in concert.

ii. **Consistency -** concerns whether the operations performed upon a collection of replicated objects produce results that meet the specification of correctness for those objects.

**System model and the role of group communication**

The data in the system consist of a collection of items called as objects that could be a file, or a Java object . Each such logical object is implemented by a collection of physical copies called replicas.

The replicas are physical objects, each stored at a single computer, with data and behaviour that are tied to some degree of consistency by the system's operation. The 'replicas' of a given object are not necessarily identical, at least not at any particular point in time.

**System Model**

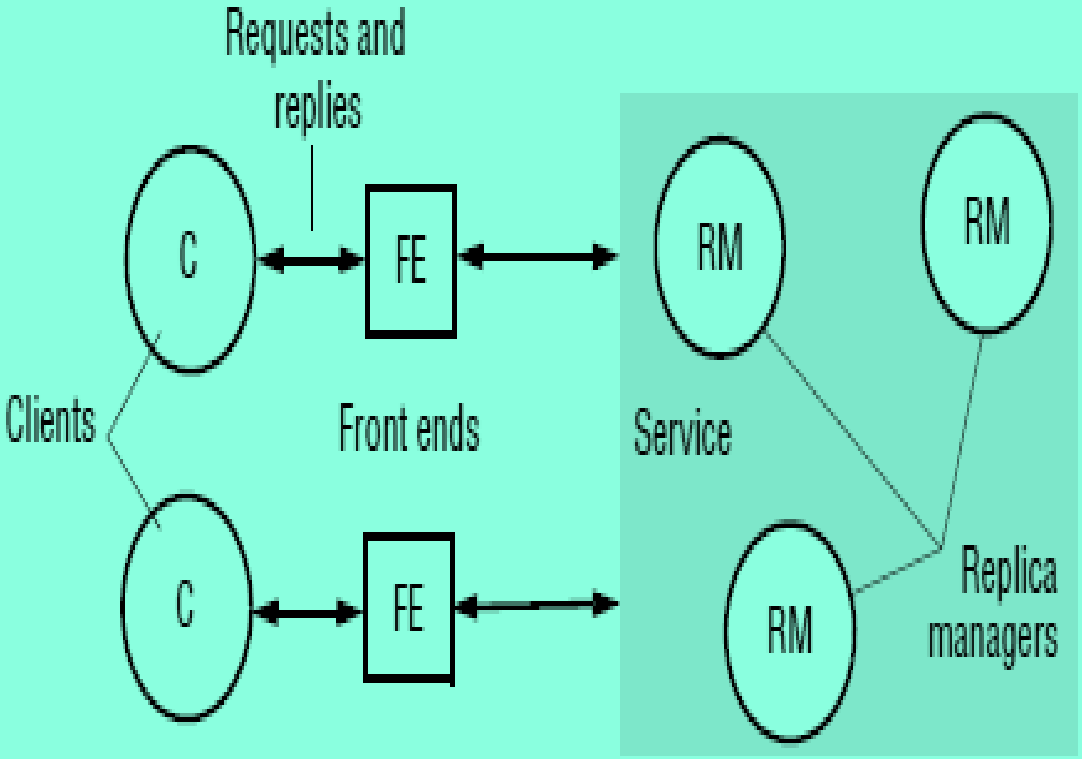
**Assumptions made:**

- ✓System is asynchronous in which processes may fail only by crashing
- ✓Network partitions may not occur. If they do occur, they make it harder to build failure detectors. Failure detectors are used to achieve reliable and totally ordered multicast.

The model involves replicas held by distinct **replica managers** that are components that contain the replicas on a given computer and perform operations directly upon them .

System Model:

Figure 18.1 A basic architectural model for the management of replicated data



A replica manager applies operations to its replicas recoverably. An operation at a replica manager does not leave inconsistent results if it fails part way through. Sometimes each replica act as a **state machine.**

Such a replica manager applies operations to its replicas atomically (indivisibly), so that its execution is equivalent to performing operations in some strict sequence. Moreover, the state of its replicas is a deterministic function of their initial states and the sequence of operations that it applies to them.

The replicas of different objects may be maintained by different sets of replica manager. The set of replica managers may be static or dynamic.

In a dynamic system, new replica managers may appear this is not allowed in a static system. In a dynamic system, replica managers may crash, and they are then deemed to have left the system. In a static system, replica managers do not crash but they may cease operating for an indefinite period.

A collection of replica managers provides a service to clients. The clients see a service that gives them access to objects which in fact are replicated at the

managers. Each client requests a series of operations – invocations upon one or more of the objects.

An operation may involve a combination of reads of objects and updates to objects. Requested operations that involve no updates are called **readonly requests**. Requested operations that update an object are called **update requests.**

Each client's requests are first handled by a component called a **front end**. The role of the front end is to communicate by message passing with one or more of the replica managers, rather than forcing the client to do this itself explicitly. It is the vehicle for making replication transparent. A front end may be implemented in the client's address space, or it may be a separate process.

In general, five phases are involved in the performance of a single request upon the replicated objects.

i. **Request:** The front end issues the request to one or more replica managers.
   – either the front end communicates with a single replica manager, which in turn communicates with other replica managers;
   – or the front end multicasts the request to the replica managers.

ii. **Coordination:** The replica managers coordinate in preparation for executing the request consistently. They also decide on the ordering of this request relative to others.

**Types of ordering:**

- ✓ **FIFO ordering**: If a front end issues request r and then request $r^1$, any correct replica manager that handles $r^1$ handles r before it.

- ✓ **Causal ordering**: If the issue of request r happened-before the issue of request $r^1$ then any correct replica manager that handles $r^1$ handles r before it.

- ✓ **Total ordering**: If a correct replica manager handles r before request $r^1$, then any correct replica manager that handles $r^1$ handles r before it.

**iii. Execution:** The replica managers execute the request – perhaps tentatively: that is, in such a way that they can undo its effects later.

**iv. Agreement**: The replica managers reach consensus on the effect of the request – if any – that will be committed. For example, in a transactional system the replica managers may collectively agree to abort or commit the transaction at this stage.

**v. Response:** One or more replica managers responds to the front end. In some systems, one replica manager sends the response. In others, the front end receives responses from a collection of replica managers and selects or synthesizes a single response to pass back to the client.

## Role of group communication

In replication there is a strong requirement for dynamic membership, in which processes join and leave the group as the system executes. Systems require more advanced features of failure detection and notification of membership changes as processes join, leave and crash.

A full group membership service maintains **group views**, which are lists of the current group members, identified by their unique process identifiers. The list is ordered, for example, according to the sequence in which the members joined the group.

A new group view is generated each time that a process is added or excluded. Group membership service may exclude a **suspected process** from a group , even though it may not have crashed. If the suspected process becomes connected again that process will have to rejoin the group.

A false suspicion of a process and the consequent exclusion of the process from the group may reduce the group's effectiveness.

**Design challenge** :

- ✓ Designing failure detectors to be accurate
- ✓ Ensuring that a system based on group communication does not behave incorrectly if a process is falsely suspected .

## Treating network partitions

Disconnection or the failure of components such as a router in a network may split a group of processes into two or more subgroups, with communication between the subgroups impossible. Group management services differ in whether they are **primary partition or partitionable.**

In **primary partition,** the management service allows at most one subgroup (a majority) to survive a partition; the remaining processes are informed that they should suspend operations. In a **partitionable group**, in some circumstances it is acceptable for two or more subgroups to continue to operate.

## View delivery:

For each group g the group management service delivers to any member process $p \varepsilon g$, a series of views v0(g), v1(g) , v2(g) , etc. For example, a series of views could be v0(g) = (p) , v1(g) = (p, $p^1$) and v2(g) = (p) – p joins an empty group, then $p^1$ joins the group, then $p^1$ leaves it.

Although several membership changes may occur concurrently, such as when one process joins the group just as another leaves, the system imposes an order on the sequence of views given to each process.

**Basic requirements for view delivery:**

**i.Order**: If a process p delivers view v(g) and then view $v^1(g)$, then no other process q $\neq$p delivers $v^1$ (g) before v(g).

**ii.Integrity**: If process p delivers view v(g), then p $\varepsilon$ v(g).

**iii. Non-triviality**: If process q joins a group and is or becomes indefinitely reachable from process p $\neq$ q, then eventually q is always in the views that p delivers.

The first of these requirements goes some way to giving the programmer a consistency guarantee by ensuring that view changes always occur in the same order at different processes. The second requirement is a ‘sanity check’. The third guards against trivial solutions.

**View-synchronous group communication**

A view-synchronous group communication system makes guarantees to the delivery ordering of view notifications.

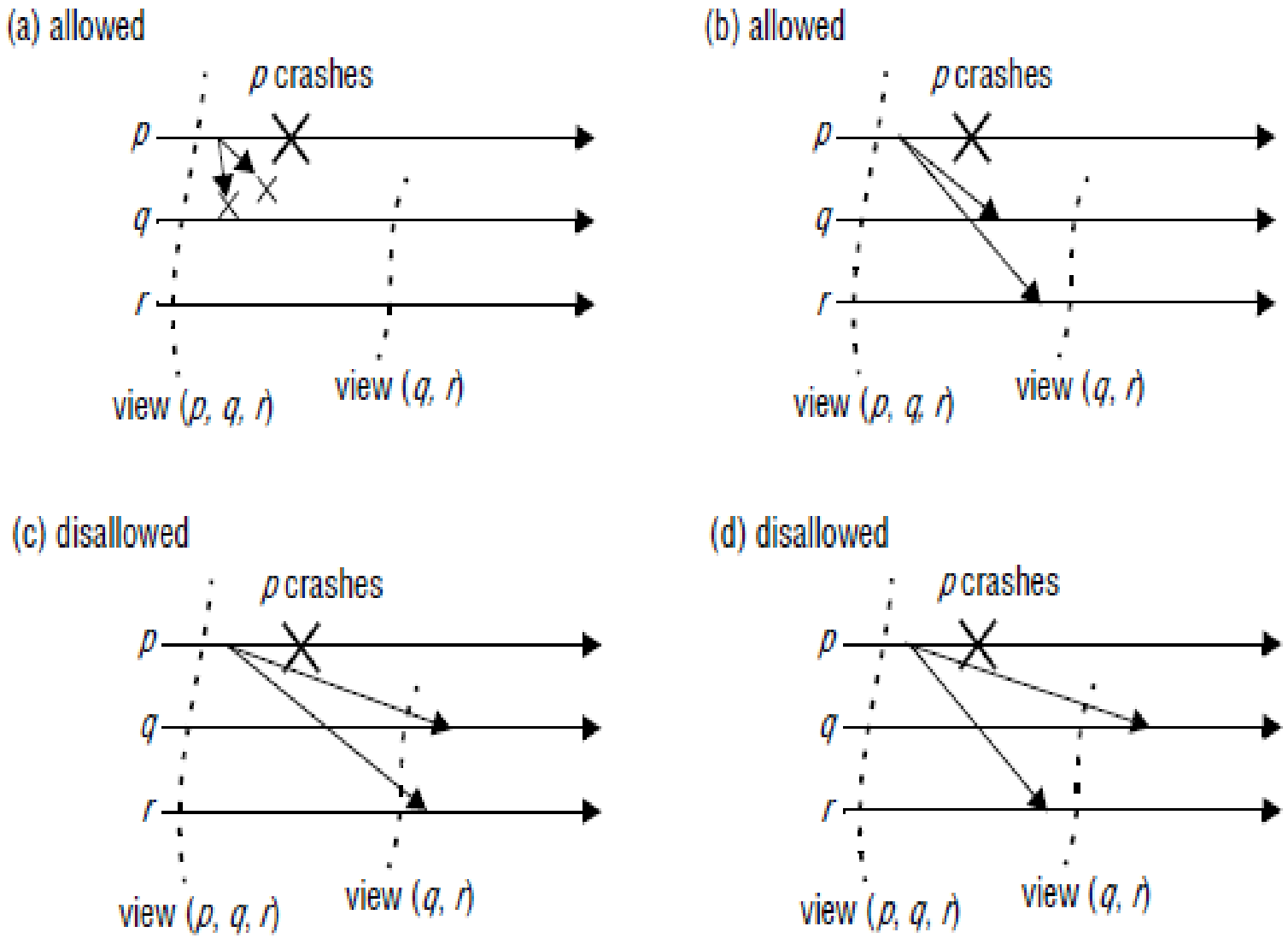
**Guarantees provided:**

i. **Agreement -** Correct processes deliver the same sequence of views and the same set of messages in any given view.

ii. **Integrity -** If a correct process p delivers message m, then it will not deliver m again.

iii. **Validity -** Correct processes always deliver the messages that they send. If the system fails to deliver a message to any process q, then it notifies the surviving processes by delivering a new view with q excluded, immediately after the view in which any of them delivered the message.

**Figure 18.2** View-synchronous group communication



Consider a group with three processes, *p*, *q* and *r* Suppose that *p* sends a message *m* while in view (*p*, *q*, *r*) but that *p* crashes soon after sending *m*, while *q* and *r* are correct.

One possibility is that *p* crashes before *m* has reached any other process. In this case, *q* and *r* each deliver the new view (*q*, *r*), but neither ever delivers *m*.

The other possibility is that *m* has reached at least one of the two surviving processes when *p* crashes. Then *q* and *r* both deliver first *m* and then the view (*q*, *r*). It is not allowed for *q* and *r* to deliver first the view (*q*, *r*) and then *m*, since then they would deliver a message from a process that they have been informed has failed; nor can the two deliver the message and the new view in opposite orders.

View-synchronous communication can be used to achieve ***state transfer***

Upon delivery of the first view containing the new process, some distinct process from amongst the pre-existing members – say, the oldest – captures its state, sends it one-to-one to the new member and suspends its execution.

All other pre-existing processes suspend their execution. Upon receipt of the state, the new process integrates it and multicasts a 'commence!' message to the group, at which point all proceed once more

## Fault-tolerant services

## Linearizability and sequential consistency

There are various correctness criteria for replicated objects. The most strictly correct systems are *linearizable*, and this property is called *linearizability*

A replicated shared object service is said to be linearizable if *for any execution* there is some interleaving of the series of operations issued by all the clients that satisfies the following two criteria

1. The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
2. The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

A weaker correctness condition is *sequential consistency*, which captures an essential requirement concerning the order in which requests are processed without appealing to real time

A replicated shared object service is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the clients that satisfies the following two criteria.

Every linearizable service is also sequentially consistent, since real-time order reflects each client's program order. The converse does not hold.

## Models for fault tolerance:

### 1. Passive Replication

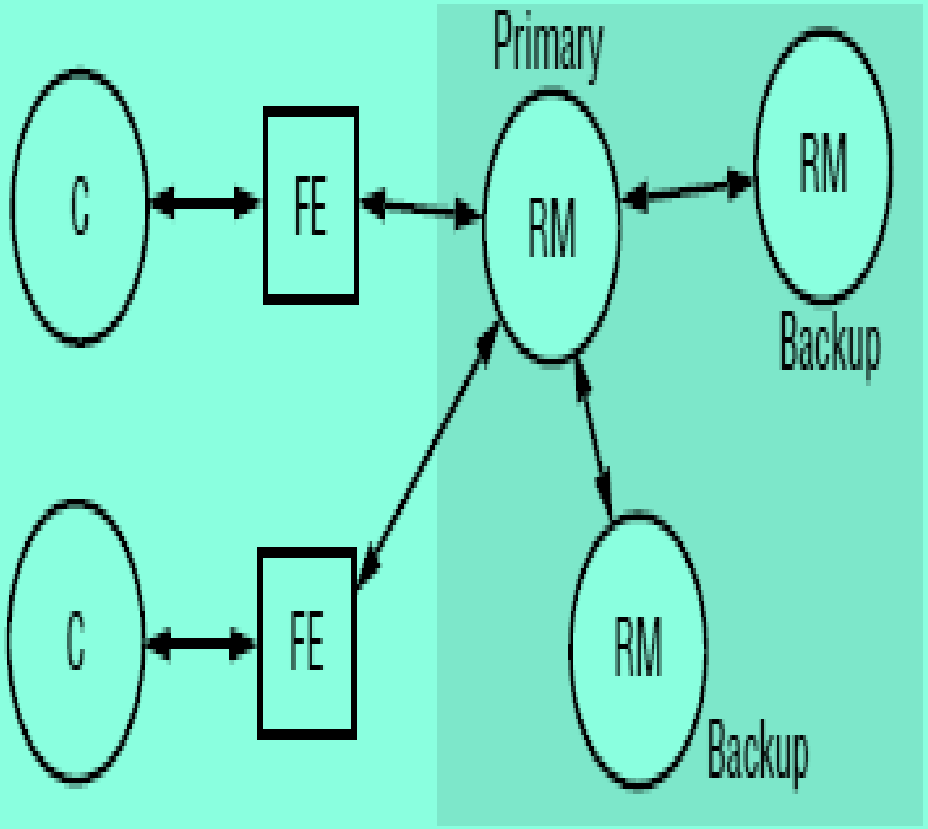
In the *passive* or *primary-backup* model of replication for fault tolerance there is at any one time a single primary replica manager and one or more secondary replica managers – ‘backups’ or ‘slaves’.

In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service.

The primary replica manager executes the operations and sends copies of the updated data to the backups.

If the primary fails, one of the backups is promoted to act as the primary

Figure 18.3 The passive (primary-backup) model for fault tolerance



The sequence of events when a client requests an operation to be performed is as follows:

1. **Request:** The front end issues the request, containing a unique identifier, to the primary replica manager.

2. **Coordination:** The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request, and if so it simply resends the response.

3. **Execution**: The primary executes the request and stores the response.

4. **Agreement**: If the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.

5. **Response**: The primary responds to the front end, which hands the response back to the client.

**Requirements:**

The primary is replaced by a unique backup. The replica managers that survive agree on which operations had been performed at the point when the replacement primary takes over.

**Working:**

Consider a front end that has not received a response. The front end retransmits the request to whichever backup takes over as the primary. The primary may have crashed at any point during the operation.

If it crashed before the agreement stage (4), then the surviving replica managers cannot have processed the request. If it crashed during the agreement stage, then they may have processed the request.

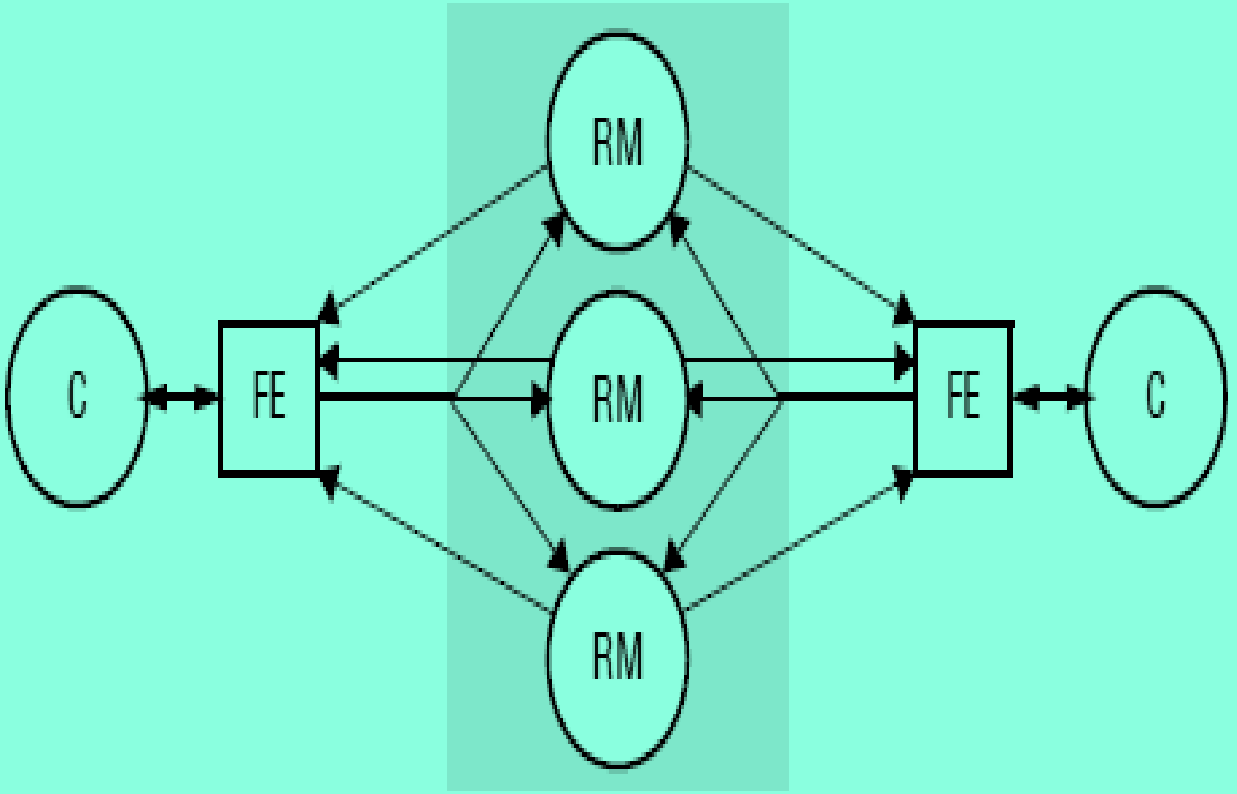
If it crashed after that stage, then they have definitely processed it. But the new primary does not have to know what stage the old primary was in when it crashed. When it receives a request, it proceeds from stage 2 above.

This system obviously implements linearizability. To survive up to $f$ process crashes, a passive replication system requires $f$ +1 replica managers (such a system cannot tolerate Byzantine failures). The front end requires little functionality to achieve fault tolerance. It just needs to be able to look up the new primary when the current primary does not respond.

Passive replication has the disadvantage of providing relatively large overheads.
**Ex**. Harp replicated file system , The Sun Network Information Service

**Active replication :**

Figure 18.4 Active replication



In the *active* model of replication for fault tolerance the replica managers are state machines that play equivalent roles and are organized as a group.

Front ends multicast their requests to the group of replica managers and all the replica managers process the request independently but identically and reply.

If any replica manager crashes, this need have no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way

The front end can collect and compare the replies it receives. Under active replication, the sequence of events when a client requests an operation to be performed is as follows:

1. **Request**: The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst. It does not issue the next request until it has received a response.

2. **Coordination**: The group communication system delivers the request to every correct replica manager in the same (total) order.

3. **Execution**: Every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.

4. **Agreement**: No agreement phase is needed, because of the multicast delivery semantics.

5. **Response**: Each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest

This system achieves sequential consistency. All correct replica managers process the same sequence of requests. The reliability of the multicast ensures that every correct replica manager processes the same set of requests and the total order ensures that they process them in the same order. Since they are state machines, they all end up with the same state as one another after each request. Each front end's requests are served in FIFO order which is the same as 'program order'.

The active replication system does not achieve linearizability. This is because the total order in which the replica managers process requests is not necessarily the same as the real-time order in which the clients made their requests.

The active replication system can mask up to $f$ Byzantine failures, as long as the service incorporates at least $2f + 1$ replica managers. Each front end waits until it has collected $f + 1$ identical responses and passes that response back to the client. It discards other responses to the same request.

To be strictly sure of which response is really associated with which request we require that the replica managers digitally sign their responses.