

# OUTLINE

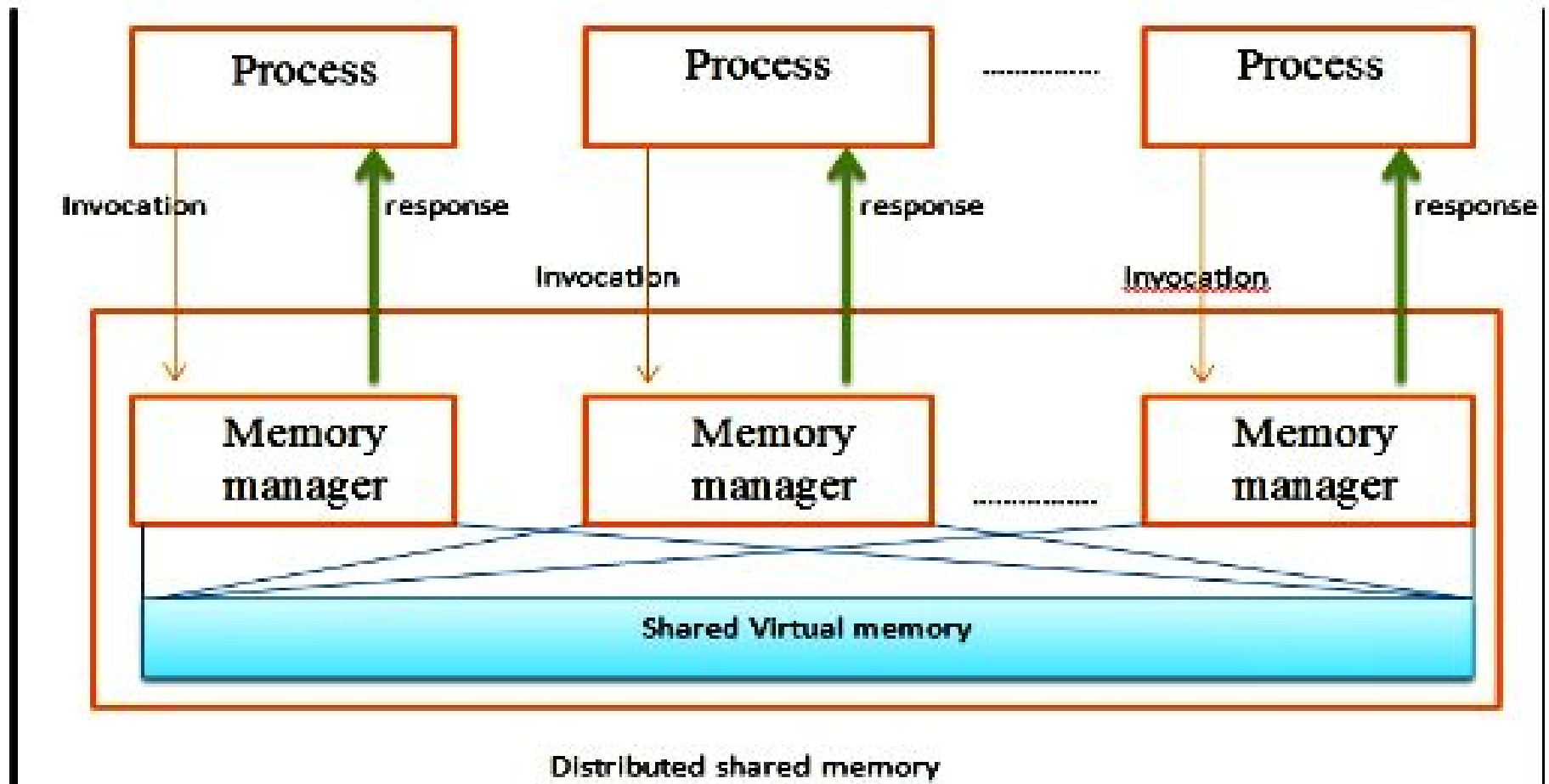
- Introduction of DSM
- Non-Uniform Memory Access Architectures
- Memory Consistency Models
- Multiprocessor Cache Systems
- Implementation of DSM Systems
- Models of Distributed Computation:
  - Preliminaries
  - Causality
  - Distributed Snapshots
  - Modelling a Distributed Computation
  - Failures in Distributed System
  - Distributed Mutual Exclusion
  - Election
  - Distributed Deadlock handling

# DISTRIBUTED SHARED MEMORY SYSTEM

- **Distributed shared memory(DSM) system** is a resource management component of **distributed operating system** that implements **shared memory** model in **distributed system** which have no physically **shared memory**. The **shared memory** model provides a virtual address space which is **shared** by all nodes in a **distributed system**.
- **Distributed shared memory (DSM)** is a form of memory architecture where physically separated memories can be addressed as one logically shared address space.
- **Note:** Here, the term "shared" does not mean that there is a single centralized memory, but that the address space is "shared" (same physical address on two processors refers to the same location in memory).

## DSM CONT...

- A distributed-memory system, often called a multicomputer, consists of multiple independent processing nodes with local memory modules which is connected by a general interconnection network.

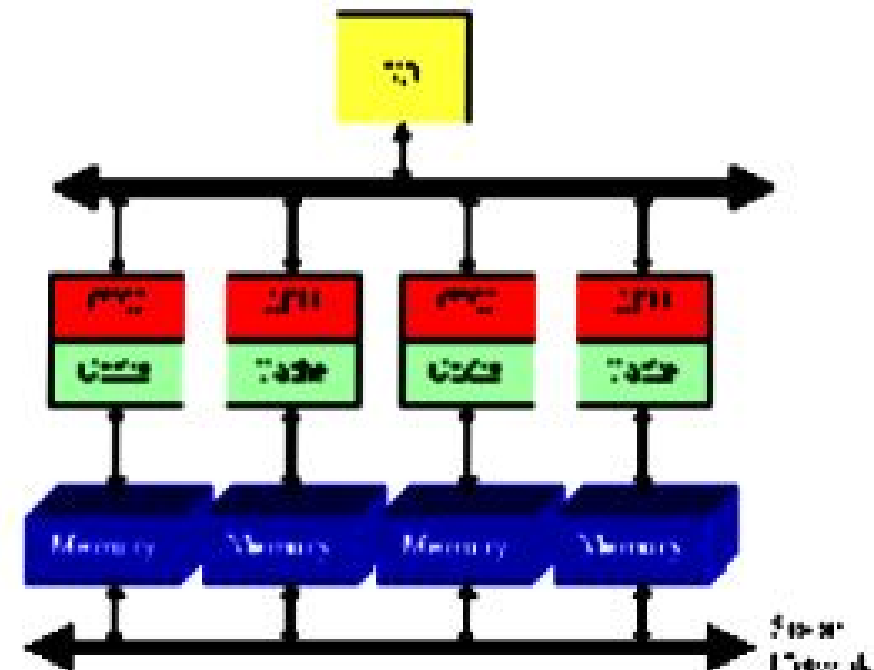


## NON-UNIFORM MEMORY ACCESS ARCHITECTURES

- ❑ NUMA is a type of parallel processing architecture.
- ❑ There are two types of parallel processing architectures – Shared Memory Architecture and Distributed Memory Architecture. Shared Memory Architectures are of two types – Uniform Memory Access (UMA) and Non Uniform Memory Access (NUMA).
- ❑ In Non Uniform Memory Access (NUMA) as shown in Figure 2, each processor has its own local memory. A processor can also have a built-in memory controller as present in Intel's Quick Path Interconnect (QPI) NUMA Architecture.
- ❑ Unlike Distributed Memory Architecture, the memory of other processor is accessible but the latency to access them is not same. The memory which is local to other processor is called as *remote memory or foreign memory*.

## NUMA CONT...

- A processor usually uses its local memory to store the data required for its processing. Accessing a local memory has least latency. It can also utilize the remote memory. Scalability is not an issue even if the count of processors grow up in this architecture.
- In Cache Coherent NUMA (ccNUMA), a processor can be directly linked via HyperTransport bus to other.



## MEMORY CONSISTENCY MODELS

- Consistency requirement vary from application to application.
- A consistency model basically refers to the degree of consistency that has to be maintained for the shared memory data.
- Defined as a set of rules that application must obey if they want the DSM system to provide the degree of consistency guaranteed by the consistency model.
- If a system support the stronger consistency model then the weaker consistency model is automatically supported but the converse is not true

## **TYPES OF MEMORY CONSISTENCY MODELS**

- ❑ **Strict Consistency model**
- ❑ **Sequential Consistency model**
- ❑ **Causal consistency model**
- ❑ **Pipelined Random Access Memory consistency model(PRAM)**
- ❑ **Processor Consistency model**
- ❑ **Weak consistency model**
- ❑ **Release consistency model**

## **STRICT CONSISTENCY MODEL**

- **This is the strongest form of memory coherence having the most stringent consistency requirement.**
- **Value returned by a read operation on a memory address is always same as the value written by the most recent write operation to that address.**
- **All writes instantaneously become visible to all processes.**
- **Implementation of the strict consistency model requires the existence of an absolute global time.**



## STRICT CONSISTENCY MODEL CONT...

- ❑ Absolute synchronization of clock of all the nodes of a distributed system is not possible.
- ❑ Implementation of strict consistency model for a DSM system is practically impossible.
- ❑ If the three operations  $\text{read}(r1)$ ,  $\text{write}(w1)$ ,  $\text{read}(r2)$  are performed on a memory location in that order.
- ❑ Only acceptable ordering for a strictly consistency memory is  $(r1, w1, r2)$

## SEQUENTIAL CONSISTENCY MODEL

- ❑ Proposed by Lamport [1979].
- ❑ A shared memory system is said to support the sequential consistency model if all processes see the same order.
- ❑ Exact order of access operations are interleaved does not matter.
- ❑ If the three operations  $\text{read}(r1)$ ,  $\text{write}(w1)$ ,  $\text{read}(r2)$  are performed on a memory location in that order.
- ❑ Any of the orderings  $(r1, w1, r2)$ ,  $(r1, r2, w1)$ ,  $(w1, r1, r2)$ ,  $(w1, r2, r1)$ ,  $(r2, r1, w1)$ ,  $(r2, w1, r1)$  is acceptable provided all processes see the same ordering

## **SEQUENTIAL CONSISTENCY MODEL CONT...**

- ❑ **The consistency requirement of the sequential consistency model is weaker than that of the strict consistency model.**
- ❑ **A sequentially consistency memory provide one-copy / single-copy semantics.**
- ❑ **Sequentially consistency is acceptable by most applications.**

## CASUAL CONSISTENCY MODEL CONT...

- ❑ Proposed by Hutto and Ahamad (1990).
- ❑ All processes see only those memory reference operations in the correct order that are potentially causally related.
- ❑ Memory reference operations not related may be seen by different processes in different order.
- ❑ Memory reference operation is said to be related to another memory reference operation if one might have been influenced by the other.
- ❑ Maintaining dependency graphs for memory access operations.

## PIPELINED RANDOM ACCESS MEMORY CONSISTENCY MODEL (PRAM)

- Proposed by Lipton and Sandberg (1988).
- Provides a weaker consistency semantics than the consistency model described so far.
- Ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed.
- All write operations performed by a single process are in a pipeline.

## PIPELINED RANDOM ACCESS MEMORY CONSISTENCY MODEL (PRAM) CONT...

- If  $w_{11}$  and  $w_{12}$  are two write operations performed by a process  $P_1$  in that order, and  $w_{21}$  and  $w_{22}$  are two write operations performed by a process  $P_2$  in that order.
- A process  $P_3$  may see them in the order  $[(w_{11}, w_{12}), (w_{21}, w_{22})]$  and another process  $P_4$  may see them in the order  $[(w_{21}, w_{22}), (w_{11}, w_{12})]$ .
- Simple and easy to implement and also has good performance.
- PRAM consistency all processes do not agree on the same order of memory reference operations

## PROCESSOR CONSISTENCY MODEL

- Proposed by Goodman [1989].
- Very similar to PRAM model with additional restriction of memory coherence.
- Memory coherence means that for any memory location all processes agree on the same order of all write operations performed on the same memory location (no matter by which process they are performed) are seen by all processes in the same order.
- If  $w_{12}$  and  $w_{22}$  are write operations for writing the same memory location  $x$ , all processes must see them in the same order-  $w_{12}$  before  $w_{22}$  or  $w_{22}$  before  $w_{12}$ .
- Processes  $P_3$  and  $P_4$  must see in the same order, which may be either  $[(w_{11}, w_{12}), (w_{21}, w_{22})]$  or  $[(w_{21}, w_{22}), (w_{11}, w_{12})]$ .

## READ-ONLY-WRITE MODEL

- Proposed by Dubois [1988].
- Common characteristics to many application:
  1. It is not necessary to show the change in memory done by every write operation to other processes eg. when a process executes in a critical section.
  2. Isolated accesses to shared variable are rare.
- Better performance can be achieved if consistency is enforced on a group of memory reference operations rather than on individual memory reference operations



- .....
- ❑ DSM system that support the weak consistency model uses a special variable called a synchronization variable.
  - ❑ Requirements:
    1. All accesses to synchronization variables must obey sequential consistency semantics.
    2. All previous write operations must be completed everywhere before an access to a synchronization variable is allowed.
    3. All previous accesses to synchronization variables must be completed before access to a non synchronization variable is allowed.

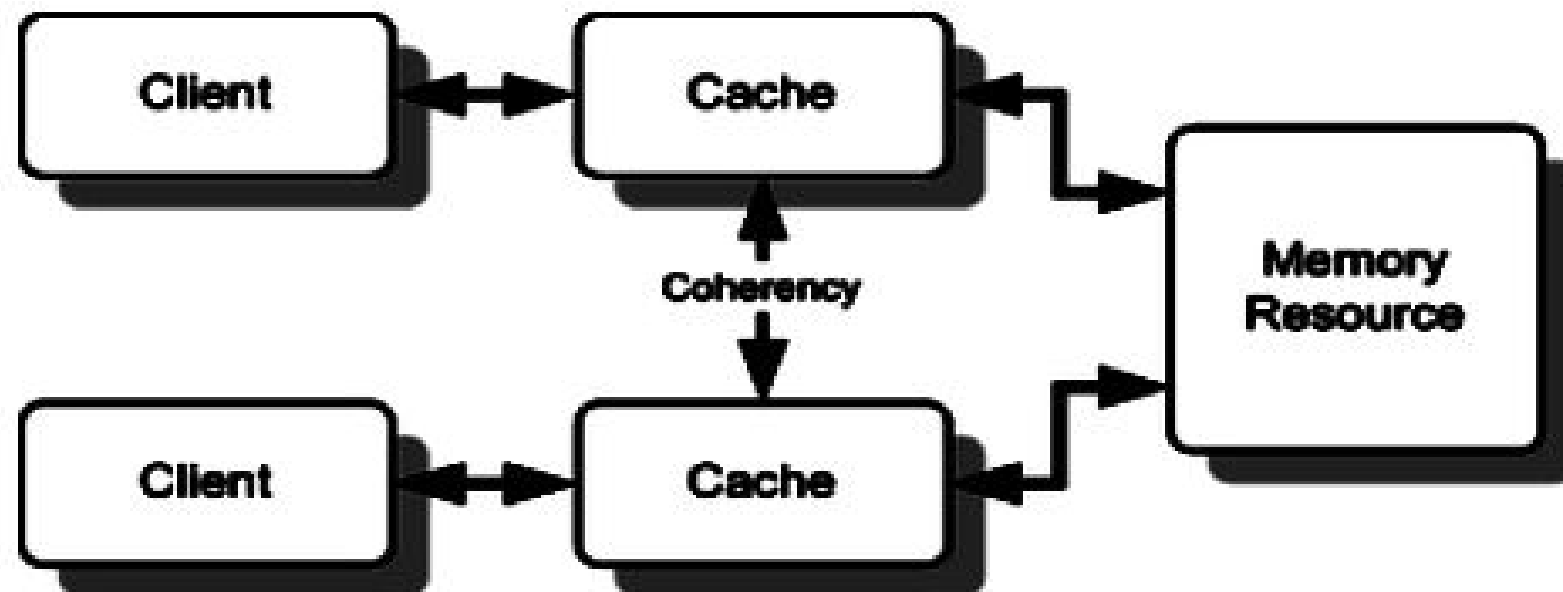
- Enhancement of weak consistency model.
- Use of two synchronization variables.
  - Acquire (used to tell the system it is entering CR).
  - Release (used to tell the system it has just exited CR).
- Acquire results in propagating changes made by other nodes to process 's node.
- Release results in propagating changes made by the process to other nodes

## RELEASE CONSISTENCY MODEL CONT...

- ▣ Barrier defines the end of a phase of execution of a group of concurrently executing processes.
- ▣ Barrier can be implemented by using a centralized barrier server .
- ▣ Requirements:
  - 1 All accesses to acquire and release synchronization variable obey processor consistency semantics.
  - 2 All previous acquires performed by a process must be completed successfully before the process is allowed to perform a data access operation on the memory.
  - 3 All previous data access operations performed by a process must be completed successfully before a release access done by the process is allowed.
- ▣ A variation of release consistency is lazy release consistency proposed by Keleher [1992]

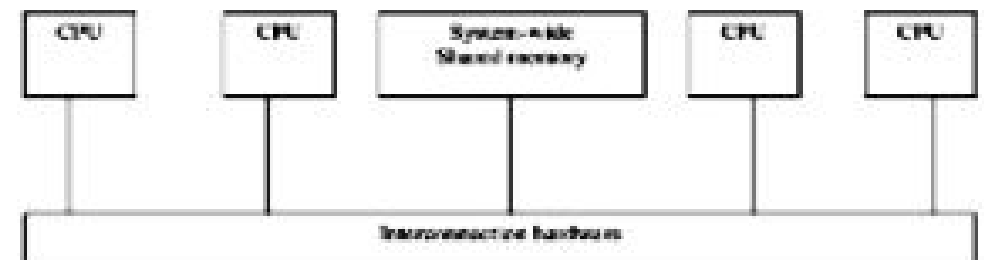
## MULTIPROCESSOR CACHE SYSTEMS

- In a shared memory **multiprocessor system** with a separate **cache** memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local **cache** of each processor that requested it.

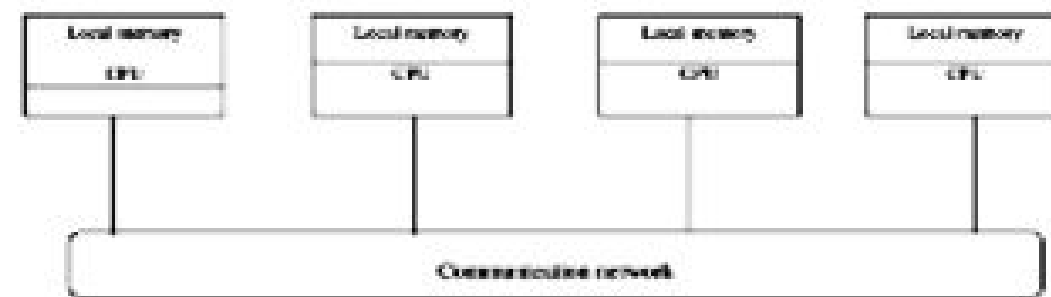


## MULTIPROCESSOR CACHE SYSTEMS CONT...

- **Multiprocessors ( tightly coupled ) :** multiple CPUs share common main memory. Shared-data approach is used.
- **Multi-computers ( loosely coupled ) :** Each CPU has its own memory and data can be shared by message-passing.



A tightly coupled multiprocessor system



A loosely coupled multiprocessor system

## IMPLEMENTATION OF DSM SYSTEMS

- ❑ Distributed Shared Memory abbreviated as **DSM** is the **implementation** of shared memory concept in distributed **systems**. The **DSM system** implements the shared memory models in loosely coupled **systems** that are deprived of a local physical shared memory in the **system**.
- ❑ In this type of system distributed shared memory provides a virtual memory space that is accessible by all the system (also known as **nodes**) of the distributed hierarchy.

Some common challenges that are to be kept in mind while the implementation of DSM –

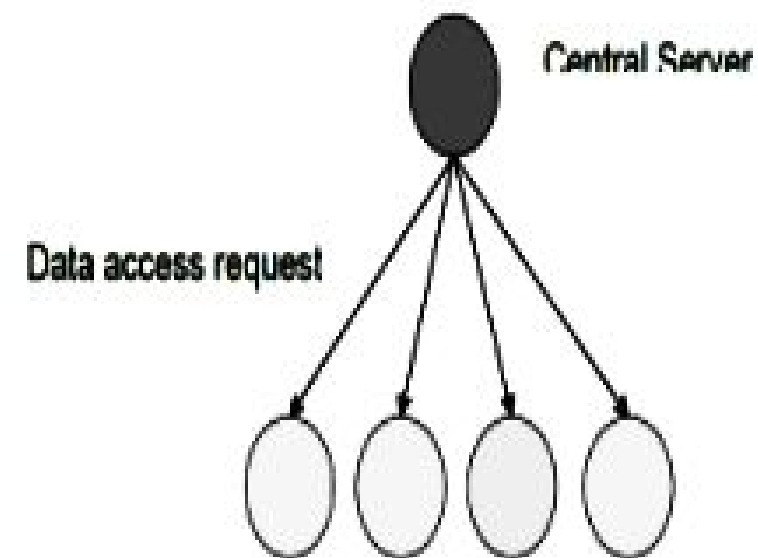
- ❑ Tracking of the memory address (location) of data stored remotely in shared memory.
- ❑ To reduce the communication delays and high overhead associated with the references to remote data.
- ❑ Controlling the concurrent access of the data shared in DSM.

Based on these challenges there are algorithms designed to implement distributed shared memory. There are four algorithms –

- Central Server Algorithm
- Migration Algorithm
- Read Replication Algorithm

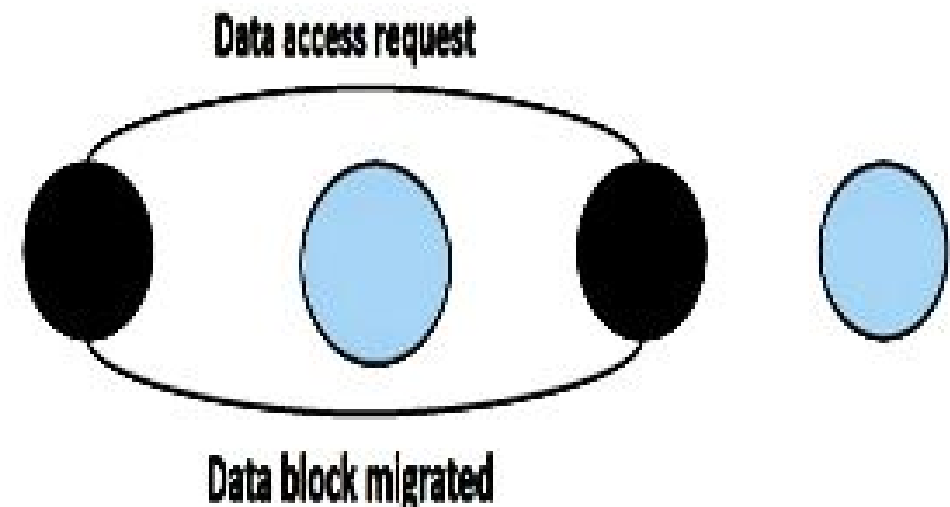
## CENTRAL SERVER ALGORITHM

- ❑ All shared data is **maintained by the central server**. Other nodes of the distributed system **request for reading and writing data** to the server which serves the request and updates or provides access to the data along with **acknowledgment messages**.
- ❑ These acknowledgment messages are used to provide the status of the data request is served by the server. When the data is sent to the calling function, it acknowledges a number that shows the access sequence of the data to maintain concurrency. And time-out is returned in case of failure.
- ❑ For larger distributed systems, there can be more than one server. In this case, the servers are located using their address or using mapping functions.



## MIGRATION ALGORITHM

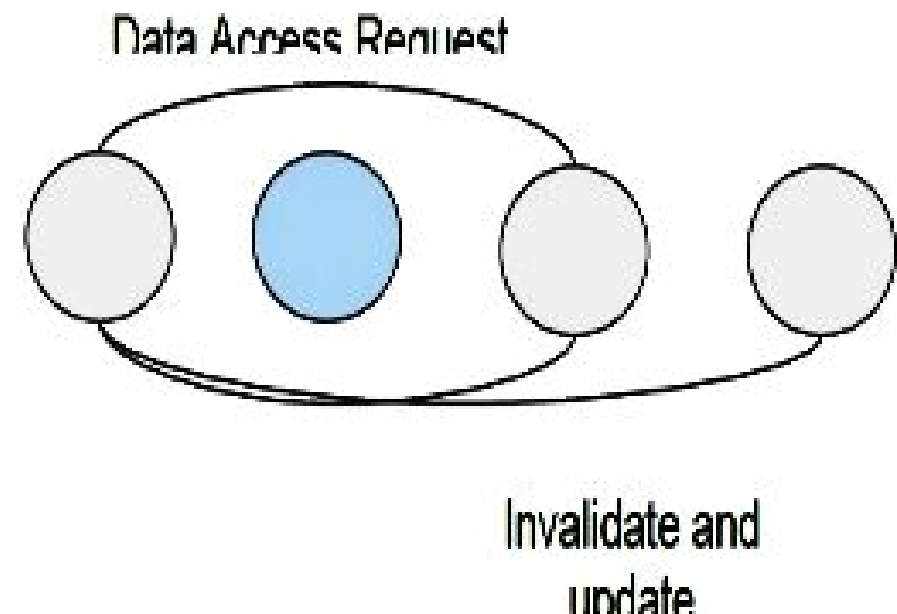
- As the name suggest the migration algorithm does the work of migration of data elements. Instead of using a central server serving each request, the **block containing the data requested by a system is migrated** to it for further access and processing. It migrates the data on request.
- This algorithm though is good if when a system accesses the same block of data multiple times and the **ability to integrate virtual memory** concept, has some shortcomings that are needed to be addressed.





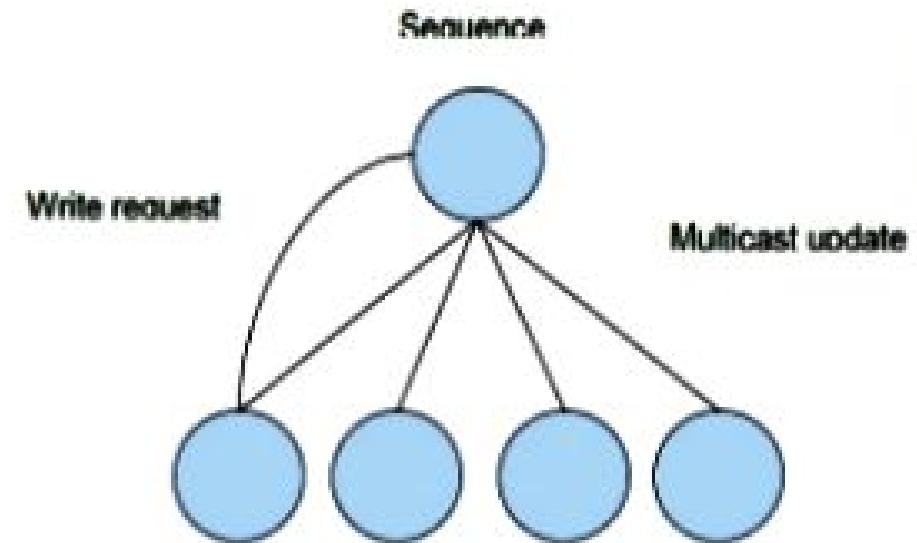
## READ REPLICATION ALGORITHM

- ❑ In the read replication algorithm, the data block that is to be accessed is **replicated** and only **reading is allowed** in all the copies. If a write operation is to be done, then all read access is put on halt till all the copies are updated.
- ❑ Overall system performance is improved as **concurrent access is allowed**. But **write operation is expensive** due to the requirement of updating all blocks that are shared to maintain concurrency. All copies of data element are to be tracked to maintain consistency.



## FULL REPLICATION ALGORITHM

- ❑ An extension to read the replication algorithm allowing the nodes to perform both **read and write** operation on the shared block of concurrently. But this access of nodes is controlled to maintain its consistency.
- ❑ To maintain consistency of data on concurrent access of all nodes sequence is maintained and after every modification that is made in the **data a multicast** with modifications is reflected all the data copies.



# MODELS OF DISTRIBUTED COMPUTATION

The model of distributed computation can be described by using points that is as follows:

- 1) Preliminaries
- 2) Causality
- 3) Distributed Snapshots
- 4) Modelling a distributed computation
- 5) Failure in DS
- 6) Distributed mutual exclusion
- 7) Election
- 8) Distributed Deadlock handling
- 9) Distributed Termination detection
- 10) Detection of dynamic termination
  - The STD algorithm
  - The DTD algorithm

# PRELIMINARIES & CAUSALITY

## 10.1 Preliminaries

Distributed protocols assume that each instance of the protocol executes on a separate processor. In general, communication is in which one processor sending a message to another. A single processor executes many tasks simultaneously and executes several instances of the same protocol. The address of a communicating entity is a combination of a machine name and the port number on the machine.

The protocol usually send control message to each other. A control message specify the action that the sending processor wishes the receiver to take and also a set of parameters. A processor can send a message of type action to processor destination via `send(destination, action, parameters)`.

In addition to sending messages, processors need to be able to receive messages as part of the protocols that they execute. The protocols will need to respond to internal interrupts and timeouts as well as external messages. The receipt of a message to the receipt of an event, which can be an external message, a timeout, or an internal interrupt. An event cannot be handled until a thread declares that it will process the event, and the event is buffered until it is handled. If the processor must be able to handle the event at any time, it must execute a thread dedicated to handling the event. A processor declares that it is waiting for events  $A_1, A_2, \dots, A_n$  by executing the following code:

```
wait for  $A_1, A_2, \dots, A_n$   
 $A_1$ (source, parameters)
```

Code to handle  $A_1$

$A_n$ (source, parameters)

Code to handle  $A_n$

When processor  $p$  executes `send( $q, A_1$ , parameters)` and processor  $q$  executes the above code,  $q$  will eventually process the message sent by  $p$ . The variable source will contain the processor name  $p$ , and the parameters that  $p$  sent will be unpacked by  $q$ . The semantics of this construction are similar to the select system call used with Berkeley sockets.

The protocols need to take an action if they suspect that a remote processor has failed. Failures are usually detected with timeouts. We use the following constructor to specify that the protocol must wait for a message of type event from  $P$  for up to  $t$  seconds and to take the specified timeout action if no such message is received:

```
wait until  $P$  send(event, parameters), timeout =  $t$   
on timeout  
timeout action
```

In the case where we want to take an action only if we received a response, and take an action on a timeout, then we use the following construction to specify the action to wait until  $P$  sends(event, parameters), timeout =  $T$

on timeout

if no timeout occurred

(successful response actions)

In the following construction, the processor waits for up to  $T$  seconds for the response. That is, if the processor finds that there are no responses to be processed at a time  $T$  seconds after the waiting started, the waiting will end and the protocol will continue. The protocol will stop waiting when all expected responses arrive.

wait up to  $T$  seconds for(event, parameters)message

event=message handling code

The concurrent threads will usually access the same set of variables, we need to describe some sort of concurrency control among the threads. Instead of using many complex lock and unlock statements in the code, we will assume that once a thread gains control of the CPU, it does not release control until it is blocked.

## 10.2 Causality

A fundamental property of distributed system is the lack of global state. There are several factors that prevent an observer from determining the global state of a distributed system.

### Noninstantaneous Communication

Suppose that two professors, Sam and Jack, stand on nearby mountain tops, mountains X and Y. Both professors have their hand on a buzzer that is loud enough to be heard on the other mountain. A third professor Jim, stand on mountain Z, which is equidistant from mountains X and Y and signals Sam and Jack to press their buzzers by lighting a fire. Jim then hears that the buzzers from the other two professors go off at the same time. However, Sam hears his own buzzer go off well before Jack. Similarly, Jack thinks that his buzzer sounded first. Thus the observer's view of the global state of the system depends on the observation point.

In a distributed system, communications are not instantaneous for many reasons, including propagation delays, contention for network resources, and lost messages that need retransmission.

# DISTRIBUTED SNAPSHOTS

A **snapshot** algorithm is used to create a consistent **snapshot** of the global state of a **distributed system**. Due to the lack of globally shared memory and a global clock, this isn't trivially possible.

A distributed snapshot algorithm captures a consistent global state of a distributed system. A global state can be described by a cut that indicates the time at which each process "checkpoints" its local state and messages. In the case of a consistent cut C (Fig), if a message crosses C, its "send" should be before C and its "receive" should be after C.

When the system is recovered from a consistent cut, every message will be sent exactly once. If a message's "send" is after C while its "receive" is before C, C becomes inconsistent (Fig 2). It will cause problems when the processes are restarted from an inconsistent cut. For example, message m2 can be executed twice in Fig.

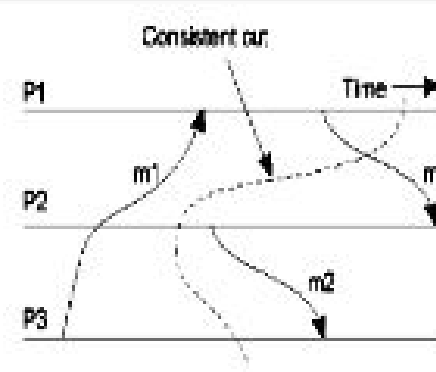


Figure 1: A Consistent Cut

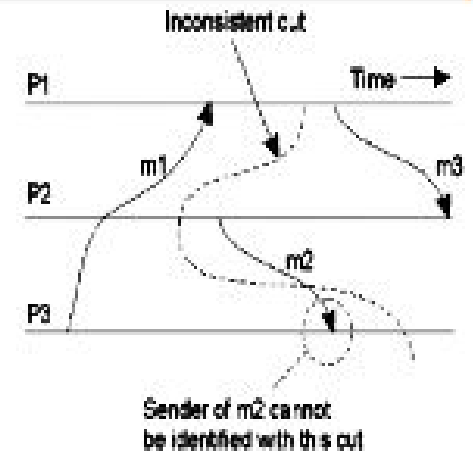


Figure 2: An inconsistent Cut

The algorithm assumes that:

- There are no network failures
- Each process communicates with another process using unidirectional point-to-point channels
- There is no message reordering on each channel
- There is a communication path between any two processes

Any process can initiate the algorithm by: 1) checkpointing its local state which contains all necessary information to restart itself, and 2) sending a marker on every outgoing channel. Then for every message from every incoming channel, the process writes a copy to disk, until a subsequent marker is received.

## DISTRIBUTED SNAPSHOTS CONT...

Now every other process may receive markers. If one process receives its first marker, it also checkpoints its local state, sends a marker on every outgoing channel and saves messages from all other incoming channels until a subsequent marker comes.

A process finishes when it receives a marker on each incoming channel. It finally collects states of all channels and send them with its own local state to initiator.

Multiple snapshots may be in progress. Each of them is separate and distinguished by tagging the marker with the initiator ID (and sequence number).

This algorithm will always stop because: 1) every process sends markers and saves incoming messages only when it receives the first marker; 2) every process stops saving messages from an incoming channel when it receives a subsequent marker from that channel, and such a marker always comes.

Intuitively, this algorithm can capture a consistent global state because: 1) no message is recorded/sent twice; 2) no message is lost, since all messages in transit are always between the two markers on the channel and therefore recorded.

## FAILURE IN DS

DSM implements **distributed systems** shared memory model in an exceedingly distributed system, that hasn't any physically shared memory.

The shared model provides a virtual address space shared between any numbers of nodes. The DSM system hides the remote communication mechanism from the appliance author, protecting the programming ease and quality typical of shared-memory systems.



## FAILURE IN DS CONT...

- Failure recovery is an interesting problem in many applications, but especially in distributed systems, where there may be multiple devices participating and multiple points of failure.
- It's very educational to identify the distinct roles in a system, and ask for each one, "What would happen if *that* part of the system failed?"
- Network failures:** participants are still running, but the connection between two or more is lost, or one or more messages are dropped before reaching the recipient. Some systems may also have issues with unexpected delays in message delivery.
- Crash failures:** a participant shuts down unexpectedly. This can occur as a result of application or environment errors, or simply a loss of power.
- Byzantine failures:** a participant may act arbitrarily. This may be due to an adversary taking control of a server, after which they may actively attempt to undermine the system. Byzantine failures remain an open research area, and are often difficult to handle unless the system was explicitly designed with potentially compromised participants in mind.
- Simultaneous or repeated failures:** these are somewhat meta-failures, in which multiple participants fail at the same time, or a single participant experiences recurring failures.
- Fault tolerant systems are those that are able to survive common failures and continue providing service even while failures are occurring.

## DISTRIBUTED MUTUAL EXCLUSION

- ❑ Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
- ❑ Only one process is allowed to execute the critical section (CS) at any given time.
- ❑ In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- ❑ Message passing is the sole means for implementing distributed mutual exclusion.

## DISTRIBUTED MUTUAL EXCLUSION CONT...

### ▣ Requirements of Mutual Exclusion Algorithms:

1 Safety Property: At any instant, only one process can execute the critical section.

2 Liveness Property: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.

3 Fairness: Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

## SOLUTIONS TO DISTRIBUTED MUTUAL EXCLUSION CONT...

- As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

### 1. Token Based Algorithm:

- A unique **token** is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence number to order requests for the critical section.
- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach insures Mutual exclusion as the token is unique

**Example:** Suzuki-Kasami's Broadcast Algorithm

## SOLUTIONS TO DISTRIBUTED MUTUAL EXCLUSION CONT...

### 2. Non-token based approach:

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme

**Example:** Lamport's algorithm, Ricart–Agrawala algorithm

## SOLUTIONS TO DISTRIBUTED MUTUAL EXCLUSION CONT...

### 3. Quorum based approach:

- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a **quorum**.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion

**Example: Maekawa's Algorithm**

## SUZUKI-KASAMI ALGORITHM

- ❑ **Suzuki-Kasami algorithm** is a token-based algorithm for achieving mutual exclusion in distributed systems. This is modification of Ricart-Agrawala algorithm, a permission based (Non-token based) algorithm which uses **REQUEST** and **REPLY** messages to ensure mutual exclusion.
- ❑ In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token. Non-token based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.
- ❑ Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.

### Data structure and Notations:

- ❑ An array of integers  $RN[1...N]$   
A site  $S_i$  keeps  $RN[1...N]$ , where  $RN[j]$  is the largest sequence number received so far through **REQUEST** message from site  $S_j$ .
- ❑ An array of integer  $LN[1...N]$   
This array is used by the token.  $LN[j]$  is the sequence number of the request that is recently executed by site  $S_j$ .
- ❑ A queue  $Q$   
This data structure is used by the token to keep record of ID of sites waiting for the token

## SUZUKI-KASAMI ALGORITHM

Algorithm:

### □ To enter Critical section:

- When a site  $S_i$  wants to enter the critical section and it does not have the token then it increments its sequence number  $RN_i[i]$  and sends a request message  $REQUEST(i, sn)$  to all other sites in order to request the token.  
Here  $sn$  is update value of  $RN_i[i]$
- When a site  $S_j$  receives the request message  $REQUEST(i, sn)$  from site  $S_i$ , it sets  $RN_j[i]$  to maximum of  $RN_j[i]$  and  $sn$  i.e  $RN_j[i] = \max(RN_j[i], sn)$ .
- After updating  $RN_j[i]$ , Site  $S_j$  sends the token to site  $S_i$  if it has token and  $RN_j[i] = LN_j[j] + 1$

### □ To execute the critical section:

- Site  $S_i$  executes the critical section if it has acquired the token.

### □ To release the critical section:

After finishing the execution Site  $S_i$  exits the critical section and does following:

- sets  $LN_i[i] = RN_i[i]$  to indicate that its critical section request  $RN_i[i]$  has been executed
- For every site  $S_j$ , whose ID is not present in the token queue  $Q$ , it appends its ID to  $Q$  if  $RN_i[j] = LN_j[j] + 1$  to indicate that site  $S_j$  has an outstanding request.
- After above updation, if the Queue  $Q$  is non-empty, it pops a site ID from the  $Q$  and sends the token to site indicated by popped ID.



## RICART AGRAWALA ALGORITHM

- ❑ **Ricart–Agrawala algorithm** is an algorithm to for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala. This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm. Like Lamport's Algorithm, it also follows permission based approach to ensure mutual exclusion.

In this algorithm:

- ❑ Two type of messages ( **REQUEST** and **REPLY**) are used and communication channels are assumed to follow FIFO order.
- ❑ A site send a **REQUEST** message to all other site to get their permission to enter critical section.
- ❑ A site send a **REPLY** message to other site to give its permission to enter the critical section.
- ❑ A timestamp is given to each critical section request using Lamport's logical clock.
- ❑ Timestamp is used to determine priority of critical section requests. Smaller time stamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

## RICART AGRAWALA ALGORITHM

### Algorithm:

#### □ To enter Critical section:

- When a site  $S_i$  wants to enter the critical section, it send a timestamped **REQUEST** message to all other sites.
- When a site  $S_j$  receives a **REQUEST** message from site  $S_i$ , It sends a **REPLY** message to site  $S_i$  if and only if
  - Site  $S_j$  is neither requesting nor currently executing the critical section.
  - In case Site  $S_j$  is requesting, the timestamp of Site  $S_i$ 's request is smaller than its own request. Otherwise the request is deferred by site  $S_j$ .

#### □ To execute the critical section:

- Site  $S_i$  enters the critical section if it has received the **REPLY** message from all other sites.

#### □ To release the critical section:

- Upon exiting site  $S_i$  sends **REPLY** message to all the deferred requests.

## MAEKAWA'S ALGORITHM

- ❑ **Maekawa's Algorithm** is quorum based approach to ensure mutual exclusion in distributed systems. As we know, In permission based algorithms like Lamport's Algorithm, Ricart-Agrawala Algorithm etc. a site request permission from every other site *but in quorum based approach, A site does not request permission from every other site but from a subset of sites which is called quorum.*

In this algorithm:

- ❑ Three type of messages ( **REQUEST**, **REPLY** and **RELEASE**) are used.
- ❑ A site send a **REQUEST** message to all other site in its request set or quorum to get their permission to enter critical section.
- ❑ A site send a **REPLY** message to requesting site to give its permission to enter the critical section.
- ❑ A site send a **RELEASE** message to all other site in its request set or quorum upon exiting the critical section.

## MAEKAWA'S ALGORITHM

### Algorithm:

#### □ To enter Critical section:

- When a site  $S_i$  wants to enter the critical section, it sends a request message **REQUEST(i)** to all other sites in the request set  $R_i$ .
- When a site  $S_j$  receives the request message **REQUEST(i)** from site  $S_i$ , it returns a **REPLY** message to site  $S_i$  if it has not sent a **REPLY** message to the site from the time it received the last **RELEASE** message. Otherwise, it queues up the request..

#### □ To execute the critical section:

- A site  $S_i$  can enter the critical section if it has received the **REPLY** message from all the site in request set  $R_i$

#### □ To release the critical section:

- When a site  $S_i$  exits the critical section, it sends **RELEASE(i)** message to all other sites in request set  $R_i$
- When a site  $S_j$  receives the **RELEASE(i)** message from site  $S_i$ , it send **REPLY** message to the next site waiting in the queue and deletes that entry from the queue
- In case queue is empty, site  $S_i$  update its status to show that it has not sent any **REPLY** message since the receipt of the last **RELEASE** message

## ELECTION

- ❑ **Distributed Algorithm** is a algorithm that runs on a distributed system. Distributed system is a collection of independent computers that do not share their memory. Each processor has its own memory and they communicate via communication networks.
- ❑ Communication in networks is implemented in a process on one machine communicating with a process on other machine.
- ❑ Many algorithms used in distributed system require a coordinator that performs functions needed by other processes in the system. **Election algorithms** are designed to choose a coordinator.

## ELECTION ALGORITHMS

- ❑ Election algorithms choose a process from group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of coordinator should be restarted.
- ❑ Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number. Then this number is send to every active process in the distributed system.

## ELECTION ALGORITHMS CONT...

- We have two election algorithms for two different configurations of distributed system.

### 1. The Bully Algorithm –

This algorithm applies to system where every process can send a message to every other process in the system.

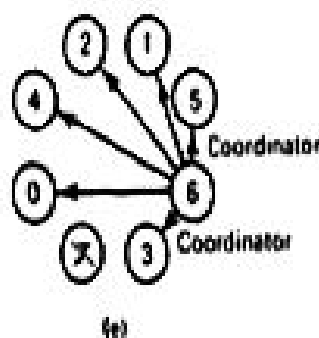
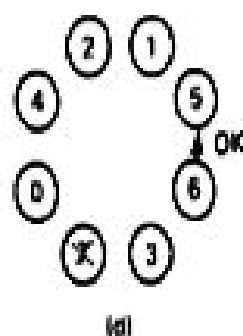
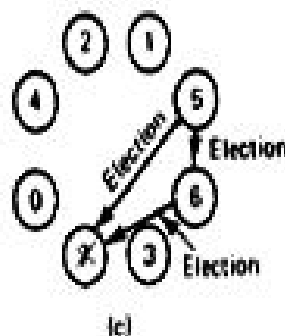
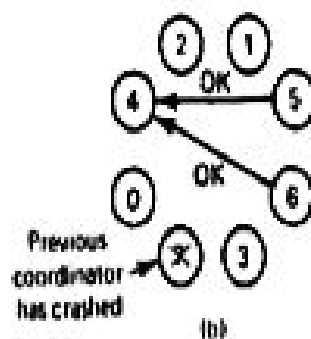
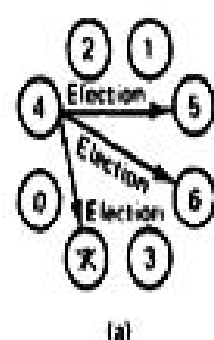
#### Algorithm –

Suppose process  $P$  sends a message to the coordinator.

- If coordinator does not respond to it within a time interval  $T$ , then it is assumed that coordinator has failed.
- Now process  $P$  sends election message to every process with high priority number.
- It waits for responses, if no one responds for time interval  $T$  then process  $P$  elects itself as a coordinator.
- Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
- However, if an answer is received within time  $T$  from any other process  $Q$ ,
  - (I) Process  $P$  again waits for time interval  $T'$  to receive another message from  $Q$  that it has been elected as coordinator.
  - (II) If  $Q$  doesn't respond within time interval  $T'$  then it is assumed to have failed

## ELECTION ALGORITHMS CONT...

◦ Example:



- In fig(a) a group of eight processes taken is numbered from 0 to 7. Assume that previously process 7 was the coordinator, but it has just crashed. Process 4 notices it first and sends ELECTION messages to all the processes higher than it that is 5, 6 and 7.
- In fig (b) processes 5 and 6 both respond with OK. Upon getting the first of these responses, process 4's job is over. It knows that one of these will become the coordinator. It just sits back and waits for the winner.
- In fig(c), both 5 and 6 hold elections by each sending messages to those processes higher than itself.
- In fig(d), process 6 tells 5 that it will take over with an OK message. At this point 6 knows that 7 is dead and that (5) it is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a COORDINATOR message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of is handled and the work can continue.
- If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.



## THE RING ALGORITHM

- ▣ This algorithm applies to systems organized as a ring(logically or physically). In this algorithm we assume that the link between the process are unidirectional and every process can message to the process on its right only. Data structure that this algorithm uses is **active list**, a list that has priority number of all active processes in the system.

### Algorithm –

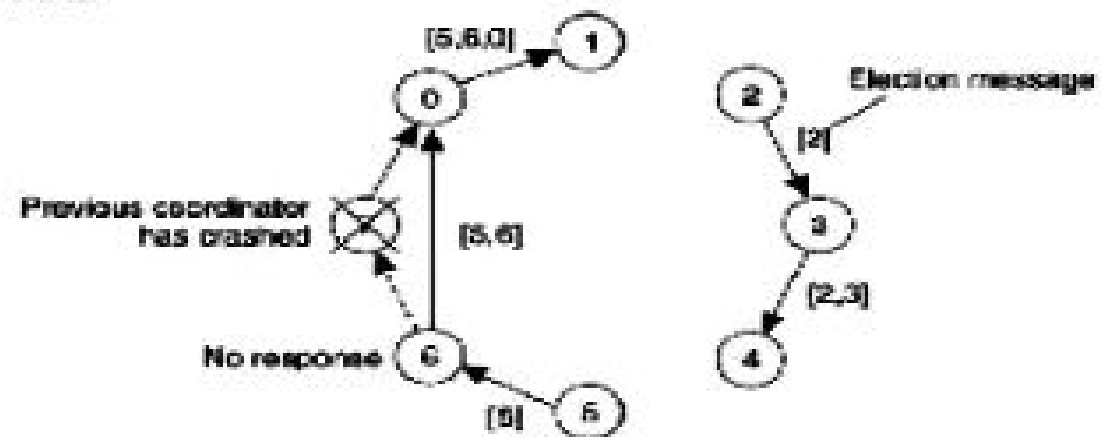
- ▣ If process P1 detects a coordinator failure, it creates new active list which is empty initially. It sends election message to its neighbour on right and adds number 1 to its active list.
- ▣ If process P2 receives message elect from processes on left, it responds in 3 ways:
  - (I) If message received does not contain 1 in active list then P1 adds 2 to its active list and forwards the message.
  - (II) If this is the first election message it has received or sent, P1 creates new active list with numbers 1 and 2. It then sends election message 1 followed by 2.
  - (III) If Process P1 receives its own election message 1 then active list for P1 now contains numbers of all the active processes in the system. Now Process P1 detects highest priority number from list and elects it as the new coordinator.

# A Ring Algorithm - Overview

- The ring algorithm assumes that the processes are arranged in a logical ring and each process knows the order of the ring of processes.
- Processes are able to “skip” faulty systems: instead of sending to process  $j$ , send to  $j + 1$ .
- Faulty systems are those that don't respond in a fixed amount of time.

# A Ring Algorithm

- P thinks the coordinator has crashed; builds an ELECTION message which contains its own ID number.
- Sends to first live successor
- Each process adds its own number and forwards to next.
- OK to have two elections at once.



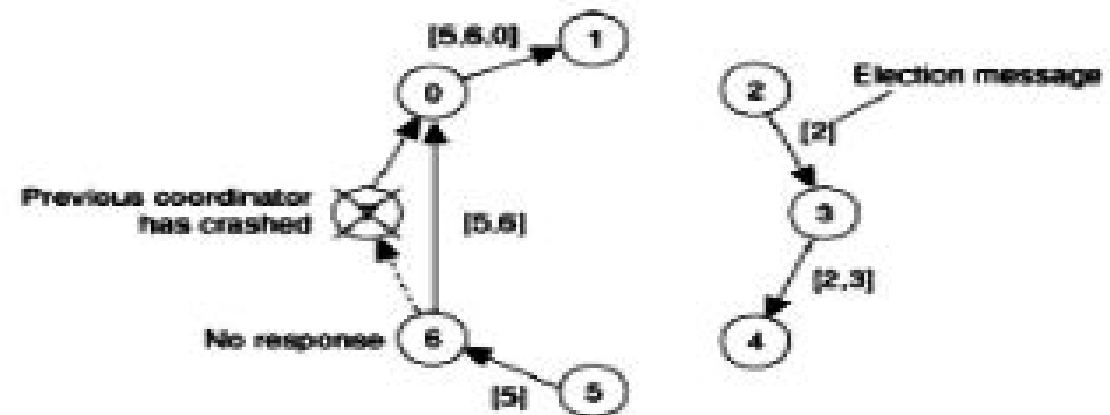
# Ring Algorithm - Details

- When the message returns to  $p$ , it sees its own process ID in the list and knows that the circuit is complete.
- $P$  circulates a COORDINATOR message with the new high number.

- Here, both 2 and 5 elect 6:

[5, **6**, 0, 1, 2, 3, 4]

[2, 3, 4, 5, **6**, 0, 1]



## DISTRIBUTED DEADLOCK HANDLING

- ❑ In a distributed system deadlock can neither be prevented nor avoided as the system is so vast that it is impossible to do so. Therefore, only deadlock detection can be implemented. The techniques of deadlock detection in the distributed system require the following:
  - ❑ **Progress –**  
The method should be able to detect all the deadlocks in the system.
  - ❑ **Safety –**  
The method should not detect false or phantom deadlocks.
- ❑ There are three approaches to detect deadlocks in distributed systems. They are as follows:

## DISTRIBUTED DEADLOCK HANDLING CONT...

### ❑ Centralized approach –

In the centralized approach, there is only one responsible resource to detect deadlock. The advantage of this approach is that it is simple and easy to implement, while the drawbacks include excessive workload at one node, single-point failure (that is the whole system is dependent on one node if that node fails the whole system crashes) which in turns makes the system less reliable.

### ❑ Distributed approach –

In the distributed approach different nodes work together to detect deadlocks. No single point failure (that is the whole system is dependent on one node if that node fails the whole system crashes) as the workload is equally divided among all nodes. The speed of deadlock detection also increases.

### ❑ Hierarchical approach –

This approach is the most advantageous. It is the combination of both centralized and distributed approaches of deadlock detection in a distributed system. In this approach, some selected nodes or clusters of nodes are responsible for deadlock detection and these selected nodes are controlled by a single node.

## DEADLOCK PREVENTION: WAIT/WOUND/DIE ALGORITHM

- ▣ Assign each process a global timestamp when it starts.  
No two processes should have same timestamp.
- ▣ **Basic idea:** "When one process is about to block waiting for a resource that another process is using, a check is made to see which has a larger timestamp (i.e. is younger)".
- ▣ *Somehow* put timestamps on each process, providing the creation time of each process. Suppose a process needs a resource already owned by another process. Determine relative ages of both processes.
- ▣ Decide if waiting process should
  - *Preempt*,
  - *Wait*,
  - *Die*, or
  - *Wound*the process that owns the resource.
- ▣ The two different algorithms by Rosenkrantz (1978) use these ideas, as explained below:

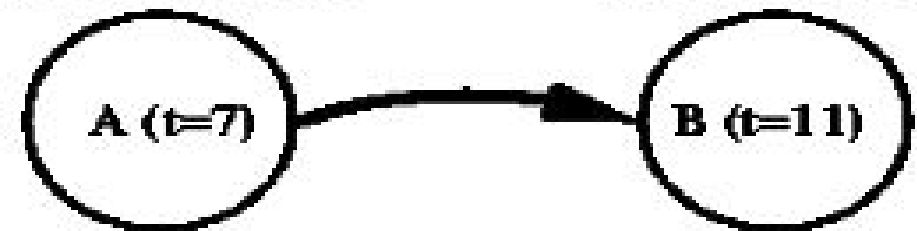
## DEADLOCK PREVENTION: WAIT/WOUND/DIE ALGORITHM

- The *Wait-Die* algorithm:  
Allow wait only if waiting process is older.  
Since timestamps increase in any chain of waiting processes, cycles are impossible.
- The Wait-Die algorithm kills the younger process. When the younger process restarts and requests the resource again, it may be killed once more. This is the less efficient of these two algorithms.

### Wait-Die Algorithm

Wants Resource

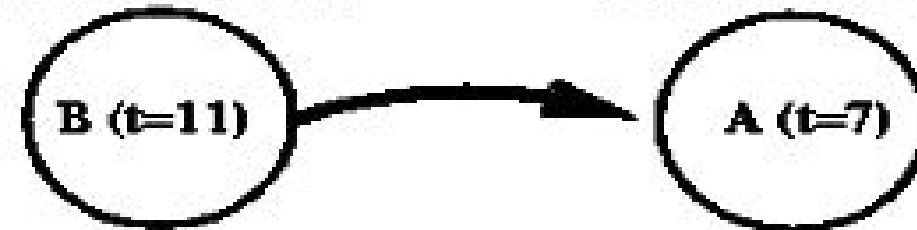
Holds Resource



*Waits*

Wants Resource

Holds Resource



*Dies*



## **DISTRIBUTED TERMINATION DETECTION**

- The basis of **termination detection** is in the concept of a **distributed system process'** state. At any time, a process in a **distributed system** is either in an active state or in an idle state.
- **Termination** occurs when all processes in the **distributed system** become idle and there are no computational messages in transit.
- **Huang's algorithm** is an algorithm for detecting termination in a distributed system. The algorithm was proposed by Shing-Tsaan Huang in 1989 in the Journal of Computers.

## HUANG'S ALGORITHM FOR TERMINATION DETECTION

- In a distributed system, a process is either in an active state or in an idle state at any given point of time. Termination occurs when all of the processes becomes idle and there are no any in transit(on its way to be delivered) computational message.

Huang's algorithm can be described by the following:

- Initially all processes are idle.
- A distributed task is started by a process sending a computational message to another process. This initial process to send the message is the "*controlling agent*".
  - The initial weight of the controlling agent is  $w$  (usually 1).
- The following rules are applied throughout the computation:
  - A process sending a message splits its current weight between itself and the message.
  - A process receiving a message adds the weight of the message to itself.
  - Upon becoming idle, a process sends a message containing its entire weight back to the controlling agent and it goes idle.
  - Termination occurs when the controlling agent has a weight of  $w$  and is in the idle state.
- Some weaknesses to Huang's algorithm are that it is unable to detect termination if a message is lost in transit or if a process fails while in an active state.

## HUANG'S ALGORITHM FOR TERMINATION DETECTION

- In a distributed system, a process is either in an active state or in an idle state at any given point of time. Termination occurs when all of the processes becomes idle and there are no any in transit(on its way to be delivered) computational message.

Huang's algorithm can be described by the following:

- Initially all processes are idle.
- A distributed task is started by a process sending a computational message to another process. This initial process to send the message is the *"controlling agent"*.
  - The initial weight of the controlling agent is  $\{\displaystyle w\}$  (usually 1).
- The following rules are applied throughout the computation:
  - A process sending a message splits its current weight between itself and the message.
  - A process receiving a message adds the weight of the message to itself.
  - Upon becoming idle, a process sends a message containing its entire weight back to the controlling agent and it goes idle.
  - Termination occurs when the controlling agent has a weight of  $\{\displaystyle w\}$  and is in the idle state.
- Some weaknesses to Huang's algorithm are that it is unable to detect termination if a message is lost in transit or if a process fails while in an active state

## DETECTION OF DYNAMIC TERMINATION

It is composed into two parts:

- 1) STD algorithm (Static)
- 2) DTD algorithm (Dynamic)

- Termination detection in distributed systems has been a popular problem of study. A large number of **termination detection** algorithms have been proposed for *static distributed systems in which the number of nodes present in the system is fixed and never changes during runtime.*
- There is relatively less work on dynamic systems, where processes may be created as well as destroyed while the computation is in progress.
- The task of termination detection in dynamic systems is more difficult because the exact number of processes participating in the computation is not known at any instant of time. Also, since processes may destroy themselves, the algorithm has to ensure that (i) the computation does not get partitioned, and (ii) a process capable of detecting termination always exists in the system. As a result, termination detection algorithms for dynamic systems are more complex than those for static systems.