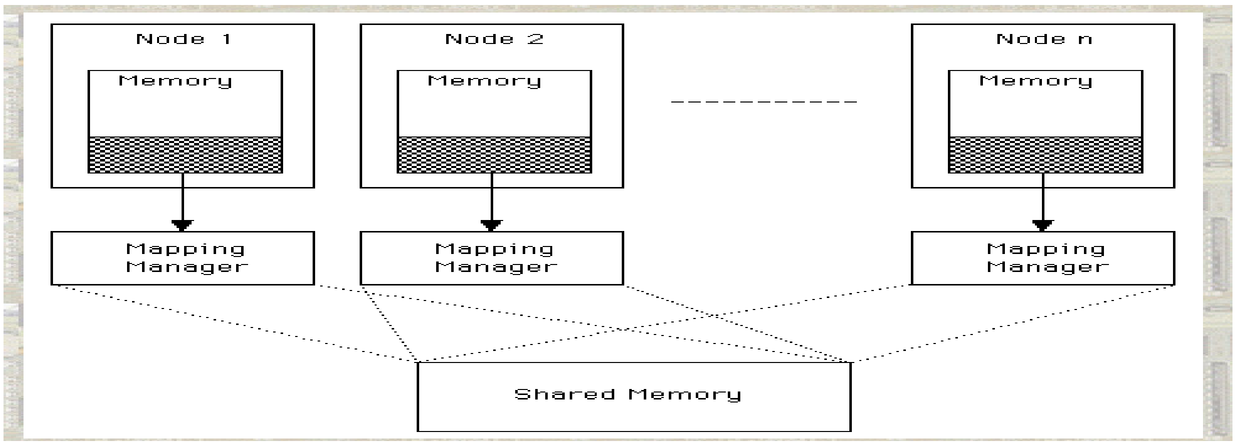
Unit 4
Distributed Shared Memory

4.1 Distributed Shared Memory

4.1.1 Introduction

Distributed Shared Memory (DSM) is a resource management component of a distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory. The shared memory model provides a virtual address space that is shared among all computers in a distributed system. An example of this layout can be seen in the following diagram taken from *Advanced Concepts in Operating Systems*.

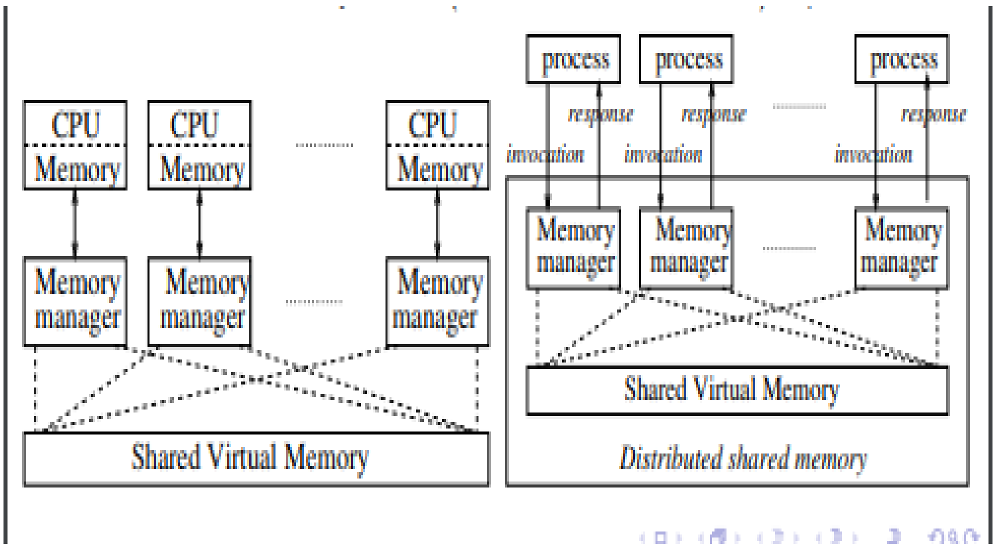


Motivation

In DSM, data is accessed from a shared address space similar to the way that virutal memory is accessed. Data moves between secondary and main memory, as well as, between the distributed main memories of different nodes. Ownership of pages in memory starts out in some pre-defined state but changes during the course of normal operation. Ownership changes take place when data moves from one node to another due to an access by a particular process.

Distributed Shared Memory Abstractions

1. communicate with Read/Write ops in shared virtual space
2. No Send and Receive primitives to be used by application I
   - Under covers, Send and Receive used by DSM manager
3. Locking is too restrictive; need concurrent access
4. With replica management, problem of consistency arises!
5. =⇒ weaker consistency models (weaker than von Neumann) reqd



Advantages/Disadvantages of DSM

Advantages:

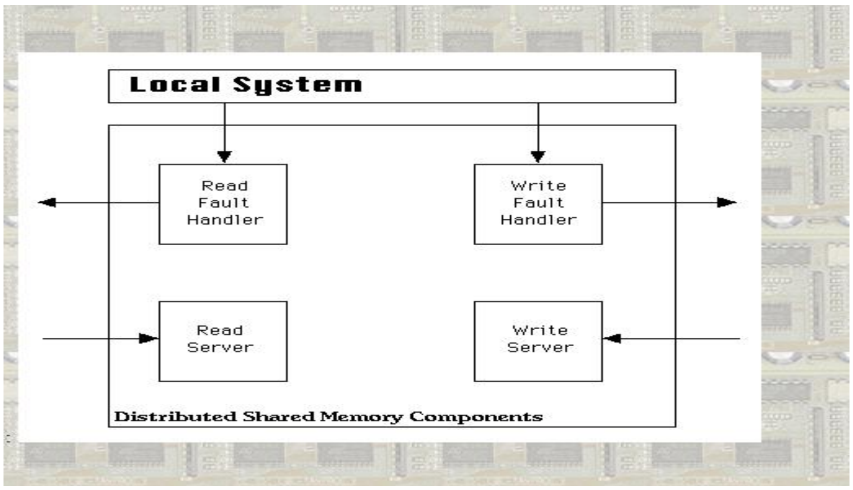
1. Shields programmer from Send/Receive primitives
2. Single address space; simpli_es passing-by-reference and passing complex data structures
3. Exploit locality-of-reference when a block is moved
4. DSM uses simpler software interfaces, and cheaper o_-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems
5. No memory access bottleneck, as no single bus Large virtual memory space
6. DSM programs portable as they use common DSM programming interface
7. Hide data movement and provide a simpler abstraction for sharing data. Programmers don't need to worry about memory transfers between machines like when using the message passing model.
8. Allows the passing of complex structures by reference, simplifying algorithm development for distributed applications.
9. Takes advantage of "locality of reference" by moving the entire page containing the data referenced rather than just the piece of data.
10. Cheaper to build than multiprocessor systems. Ideas can be implemented using normal hardware and do not require anything complex to connect the shared memory to the processors.
11. Larger memory sizes are available to programs, by combining all physical memory of all nodes. This large memory will not incur disk latency due to swapping like in traditional distributed systems.
12. Unlimited number of nodes can be used. Unlike multiprocessor systems where main memory is accessed via a common bus, thus limiting the size of the multiprocessor system.
13. Programs written for shared memory multiprocessors can be run on DSM systems,

Advantages/Disadvantages of DSM

Disadvantages:

1. Programmers need to understand consistency models, to write correct programs
2. DSM implementations use async message-passing, and hence cannot be more e_cient than msg-passing implementations
3. By yielding control to DSM manager software, programmers cannot use their own msg-passing solutions.

4.1.2 Organization



Fault Handlers

A fault handler is a proccess or potrion of a process that sits and waits for memory faults. When there is a memory access that it cannot deal with locally, the fault handler will make a request to a server on some other machine in the DSM environment. It is in charge of making sure an application or program is given the memory pages it needs without knowing what is going on underneath and where the page is actually coming from.

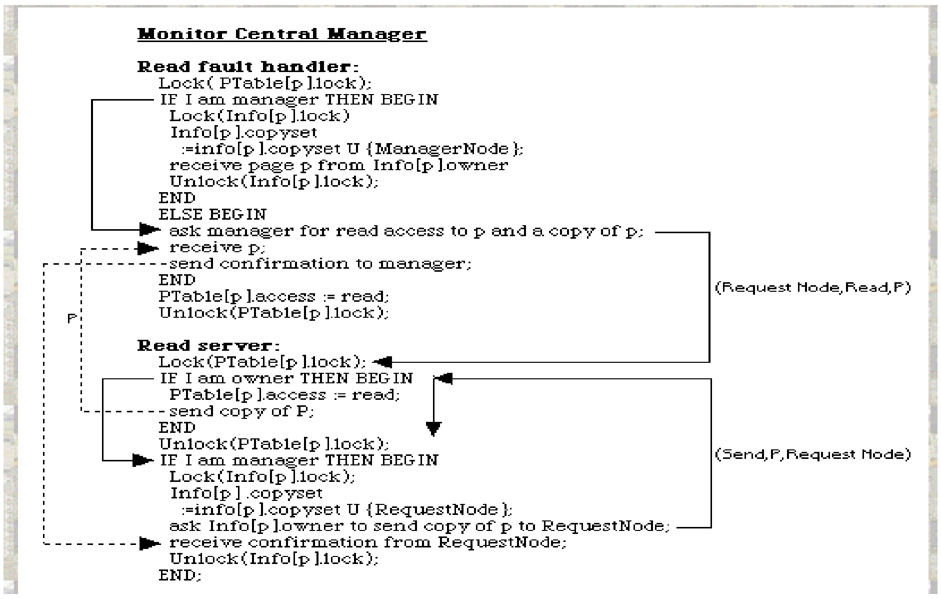
Servers

The servers from the above diagram actually service the fault handlers requests. They know which machines own the memory page that is being accessed and can fetch the page and deliver it to the asking process.

Example Code of Distributed Shared Memory Servers

Monitor Central Manager

Read Processes

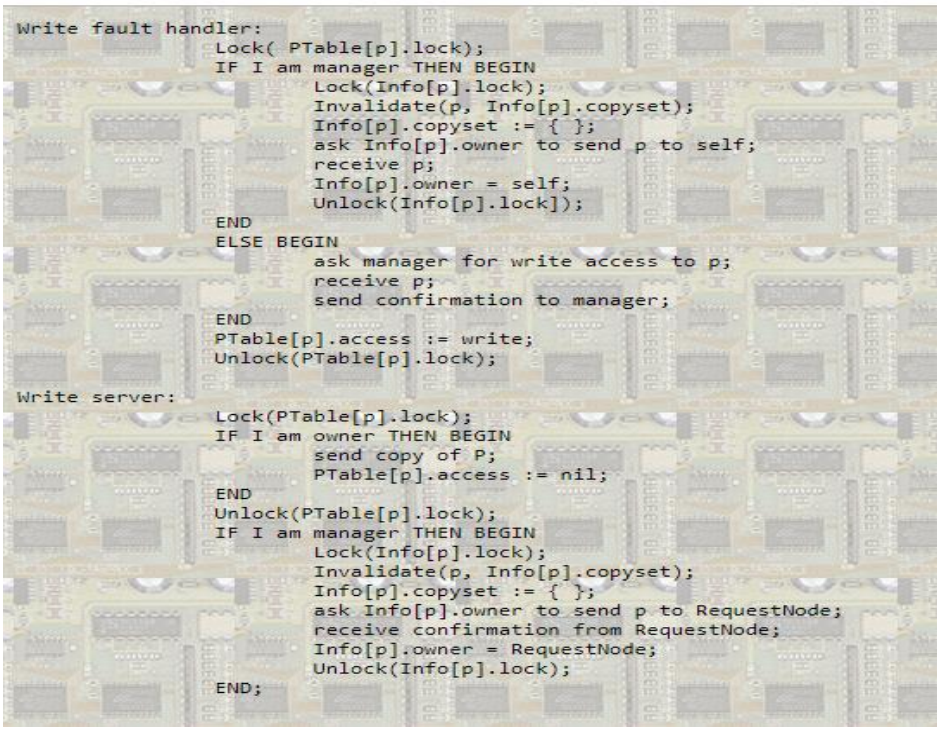


```
Monitor Central Manager

Read fault handler:
  Lock( PTable[p].lock);
  IF I am manager THEN BEGIN
    Lock(Info[p].lock)
    Info[p].copyset
      :=info[p].copyset U {ManagerNode};
    receive page p from Info[p].owner
    Unlock(Info[p].lock);
  END
  ELSE BEGIN
    ask manager for read access to p and a copy of p;
    receive p;
    send confirmation to manager;
  END
  PTable[p].access := read;
  Unlock(PTable[p].lock);

Read server:
  Lock(PTable[p].lock);
  IF I am owner THEN BEGIN
    PTable[p].access := read;
    send copy of P;
  END
  Unlock(PTable[p].lock);
  IF I am manager THEN BEGIN
    Lock(Info[p].lock);
    Info[p] .copyset
      :=info[p].copyset U {RequestNode};
    ask Info[p].owner to send copy of p to RequestNode;
    receive confirmation from RequestNode;
    Unlock(Info[p].lock);
  END;
```

Write Processes

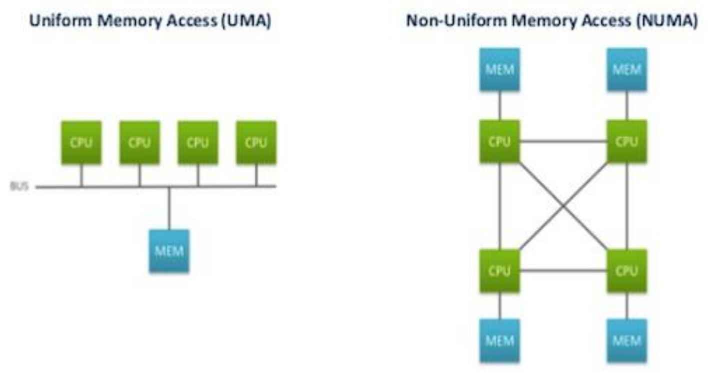


```
Write fault handler:
        Lock( PTable[p].lock);
        IF I am manager THEN BEGIN
                Lock(Info[p].lock);
                Invalidate(p, Info[p].copyset);
                Info[p].copyset := { };
                ask Info[p].owner to send p to self;
                receive p;
                Info[p].owner = self;
                Unlock(Info[p].lock]);
        END
        ELSE BEGIN
                ask manager for write access to p;
                receive p;
                send confirmation to manager;
        END
        PTable[p].access := write;
        Unlock(PTable[p].lock);
Write server:
        Lock(PTable[p].lock);
        IF I am owner THEN BEGIN
                send copy of P;
                PTable[p].access := nil;
        END
        Unlock(PTable[p].lock);
        IF I am manager THEN BEGIN
                Lock(Info[p].lock);
                Invalidate(p, Info[p].copyset);
                Info[p].copyset := { };
                ask Info[p].owner to send p to RequestNode;
                receive confirmation from RequestNode;
                Info[p].owner = RequestNode;
                Unlock(Info[p].lock);
        END;
```

4.2 Non-Uniform Memory Access Architectures

4.2.1 Introduction

Non-uniform memory access (NUMA) is a kind of memory architecture that allows a processor faster access to contents of memory than other traditional techniques. In other words, in a NUMA architecture, a processor can access local memory much faster than non-local memory. This is because in a NUMA setup, each processor is assigned a specific local memory exclusively for its own use. This eliminates sharing of non-local memory, reducing delays when multiple requests come in for access to the same memory location.

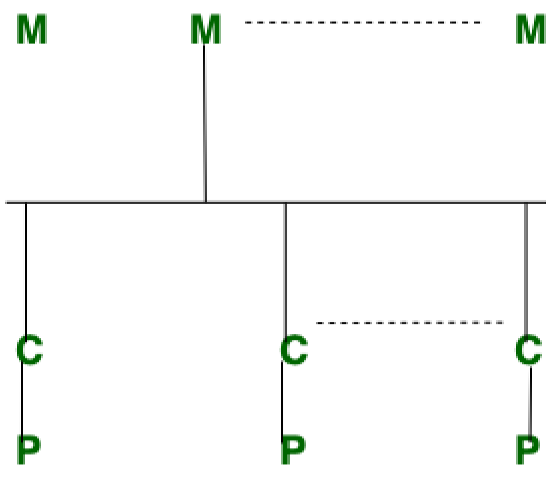


Difference between Uniform Memory Access (UMA) and Non-uniform Memory Access (NUMA)

Multiprocessors can be categorized into three shared-memory model which are:

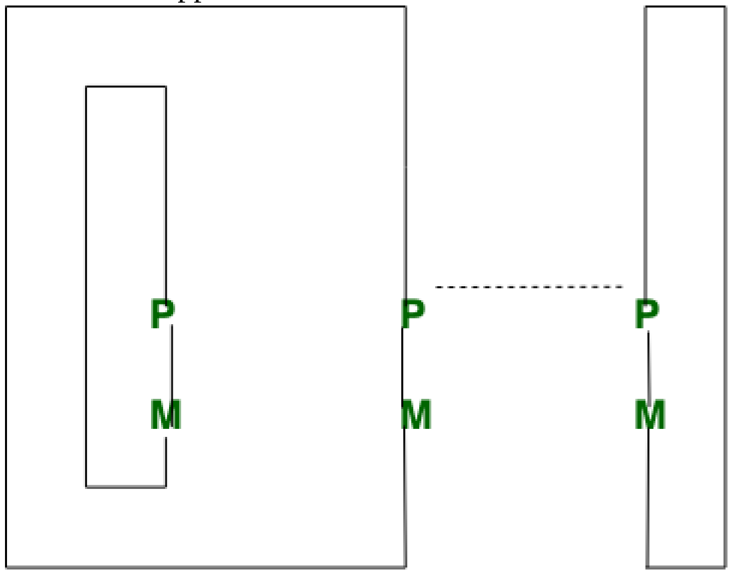
1. Uniform Memory Access (UMA)
2. Non-uniform Memory Access (NUMA)
3. Cache-only Memory Access (COMA)

Uniform Memory Access (UMA):
In UMA, where Single memory controller is used. Uniform Memory Access is slower than non-uniform Memory Access. In Uniform Memory Access, bandwidth is restricted or limited rather than non-uniform memory access. There are 3 types of buses used in uniform Memory Access which are: Single, Multiple and Crossbar. It is applicable for general purpose applications and time-sharing applications.



UMA shared memory

Non-uniform Memory Access (NUMA):
In NUMA, where different memory controller is used. Non-uniform Memory Access is faster than uniform Memory Access. Non-uniform Memory Access is applicable for real-time applications and time-critical applications.



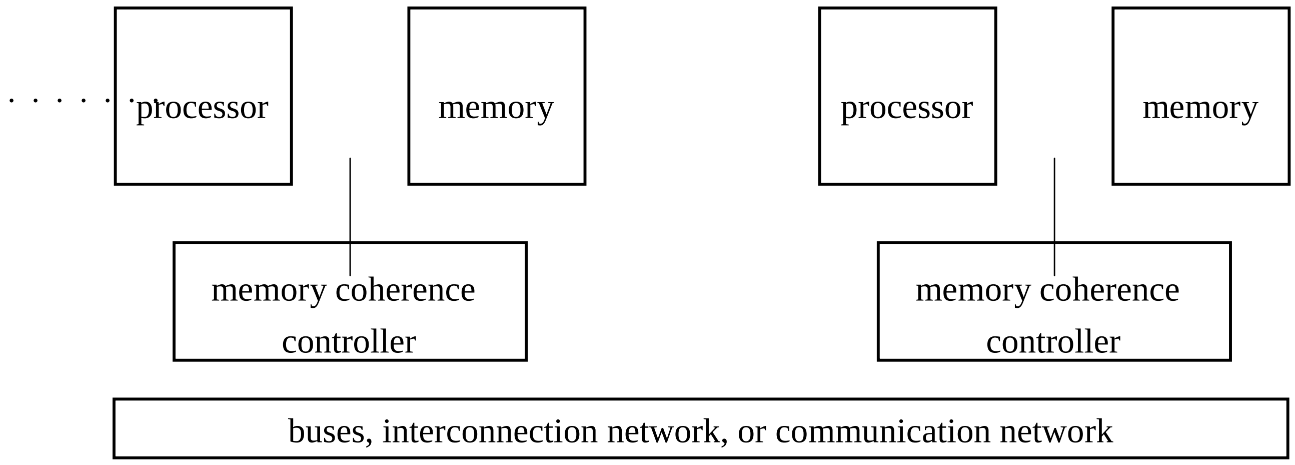
NUMA shared memory

Let’s see the difference between UMA and NUMA:

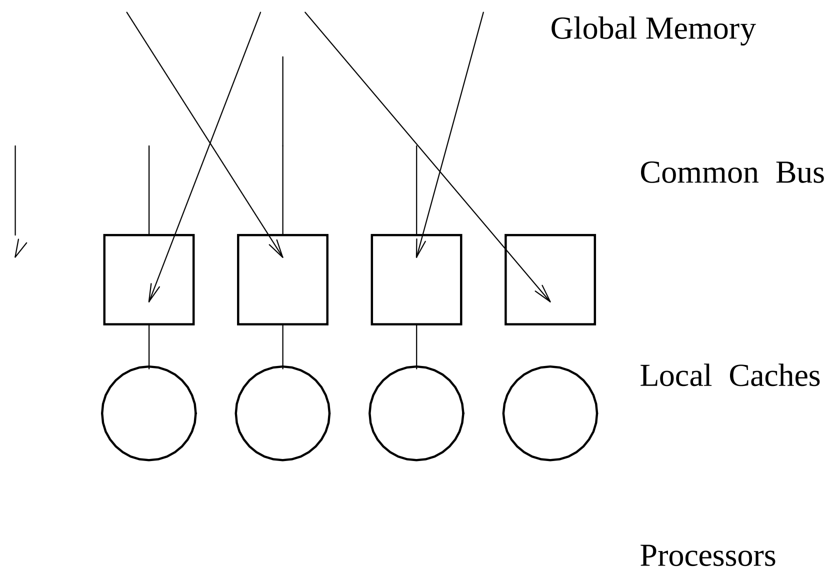
S.NO	UMA	NUMA
1.	UMA stands for Uniform Memory Access.	NUMA stands for Non-uniform Memory Access.
2.	In Uniform Memory Access, Single memory controller is used.	In Non-uniform Memory Access, Different memory controller is used.
3.	Uniform Memory Access is slower than non-uniform Memory Access.	Non-uniform Memory Access is faster than uniform Memory Access.
4.	Uniform Memory Access has limited bandwidth.	Non-uniform Memory Access has more bandwidth than uniform Memory Access.
5.	Uniform Memory Access is	Non-uniform Memory Access is

	applicable for general purpose applications and time-sharing applications.	applicable for real-time applications and time-critical applications.
6.	In uniform Memory Access, memory access time is balanced or equal.	In non-uniform Memory Access, memory access time is not equal.
7.	There are 3 types of buses used in uniform Memory Access which are: Single, Multiple and Crossbar.	While in non-uniform Memory Access, There are 2 types of buses used which are: Tree and hierarchical.

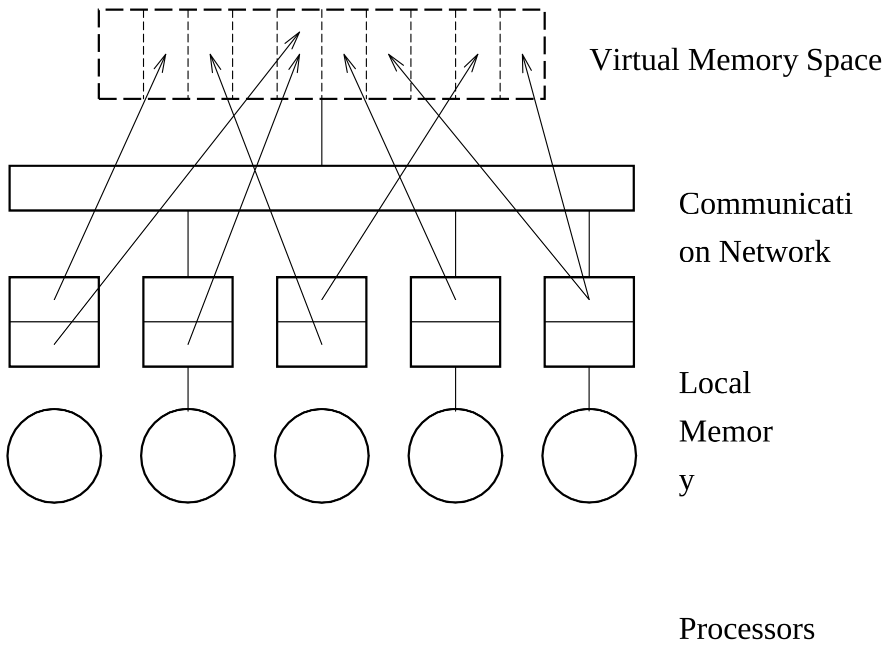
Nonuniform Memory Access (NUMA) architectures Generic NUMA architecture


. . . . . .
processor
memory
processor
memory
memory coherence controller
memory coherence controller
buses, interconnection network, or communication network

Multiprocessor Cache and DSM architectures



(a) Multiprocessor cache architecture



Distributed shared memory architecture

Common issues

Data consistency and coherency due to data placement, migration and repli- cation

- Data Sharing Granularity
- Cache Miss Granularity
- Tradeoffs:
  - Transfer time
  - Administrative overhead
  - Hit rate
  - Replacement rate
  - False Sharing

What to do on cache

miss?

- Locating block - owner/directory
- Block Migration - block bouncing
- Block Replication
- Push vs. Pull

4.3 Memory consistency models

These models apply consistency constraints to all memory accesses Accesses may require multiple messages and take significant time

Atomic consistency

All processors see same (global) order Respects real-time order

MEMORY CONSISTENCY MODELS
The memory consistency model defines the legal

ordering of memory references issued by a processor, as observed by other processors.[2,14] Different types of parallel applications inherently require various consistency models. The model's restrictiveness largely influences system performance in executing these applications. Stronger forms of the consistency model typically increase memory access latency and bandwidth requirements, while simplifying programming. Looser constraints in more relaxed models, which allow memory reordering, pipelining, and overlapping, consequently improve performance, at the expense of higher programmer involvement in synchronizing shared data accesses. For optimal behavior, systems with multiple consistency models adaptively

Hardware DSM implementations

According to the memory system architecture, three groups of hardware DSM systems are especially interesting:

- cache coherent nonuniform memory architectures (CC-NUMA),
- cache-only memory architectures (COMA), and
- reflective memory system (RMS).

CC-NUMA DSM SYSTEMS

A CC-NUMA system (see Figure B) statically distributes the shared virtual address space across local memories of clusters, which both local processors and processors from other clusters in the system can access, although with quite different access latencies. The DSM mechanism relies on directories with organization varying from a full map to different dynamic structures, such as singly or doubly linked lists and trees. The main effort is to achieve high performance (as in full-map schemes) and good scalability provided by reducing the directory storage overhead. To minimize latency, static partitioning of data should be done carefully, to maximize the frequency of local accesses. Performance indicators also depend highly on the interconnection topology. The invalidation mechanism is typically applied to provide consistency, while some relaxed memory consistency model can serve as a source of performance improvement. Typical representatives of this approach are Memnet, Dash, and SCI (see Table C.)

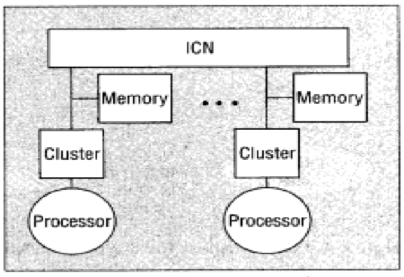
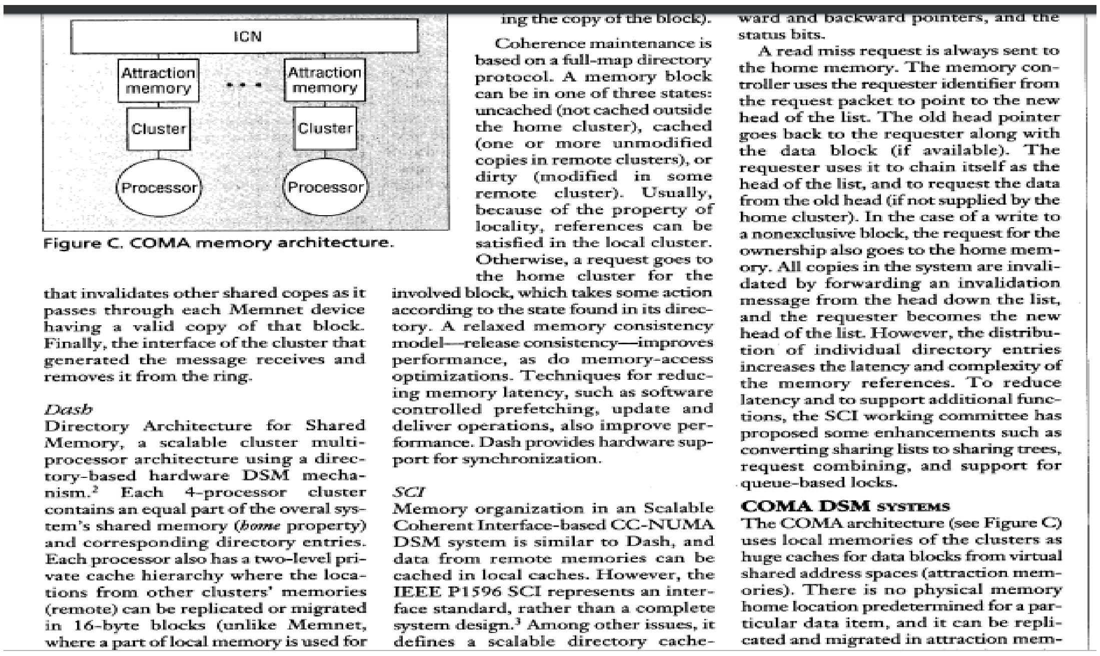


Figure B. CC-NUMA memory architecture.

Memnet

This ring-based multiprocessor—Memory as Network Abstraction—was one of the earliest hardware DSM systems.[1] The main goal was to avoid costly interprocessor communication via messages and to provide an abstraction of shared memory to an application directly by the network, without kernel OS intervention. The Memnet address space maps onto the local memories of each cluster (the reserved area) in a NUMA fashion. Another part of each local memory is the cache area, which is used for replication of 32-byte blocks whose reserved area is in some remote host. The coherence protocol is implemented in hardware state machines of the Memnet device in each cluster—a dual port memory controller on its local bus and an interface to the ring.

On a miss in local memory, the Memnet device sends an appropriate message, which circulates on the ring. Each Memnet device on the ring inspects the message in a snooping manner. The nearest cluster with a valid copy satisfies the request by inserting requested data in the message before forwarding. The write request to a nonexclusive copy results in a message



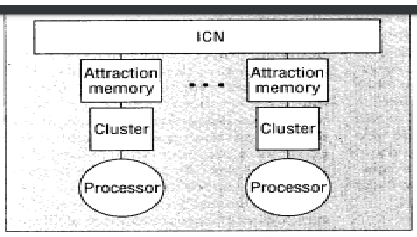


Figure C. COMA memory architecture.

that invalidates other shared copes as it passes through each Memnet device having a valid copy of that block. Finally, the interface of the cluster that generated the message receives and removes it from the ring.

Dash
Directory Architecture for Shared Memory, a scalable cluster multiprocessor architecture using a directory-based hardware DSM mechanism.[2] Each 4-processor cluster contains an equal part of the overal system's shared memory (*home* property) and corresponding directory entries. Each processor also has a two-level private cache hierarchy where the locations from other clusters' memories (remote) can be replicated or migrated in 16-byte blocks (unlike Memnet, where a part of local memory is used for

ing the copy of the block).

Coherence maintenance is based on a full-map directory protocol. A memory block can be in one of three states: uncached (not cached outside the home cluster), cached (one or more unmodified copies in remote clusters), or dirty (modified in some remote cluster). Usually, because of the property of locality, references can be satisfied in the local cluster. Otherwise, a request goes to the home cluster for the involved block, which takes some action according to the state found in its directory. A relaxed memory consistency model—release consistency—improves performance, as do memory-access optimizations. Techniques for reducing memory latency, such as software controlled prefetching, update and deliver operations, also improve performance. Dash provides hardware support for synchronization.

SCI
Memory organization in an Scalable Coherent Interface-based CC-NUMA DSM system is similar to Dash, and data from remote memories can be cached in local caches. However, the IEEE P1596 SCI represents an interface standard, rather than a complete system design.[3] Among other issues, it defines a scalable directory cache-

ward and backward pointers, and the status bits.

A read miss request is always sent to the home memory. The memory controller uses the requester identifier from the request packet to point to the new head of the list. The old head pointer goes back to the requester along with the data block (if available). The requester uses it to chain itself as the head of the list, and to request the data from the old head (if not supplied by the home cluster). In the case of a write to a nonexclusive block, the request for the ownership also goes to the home memory. All copies in the system are invalidated by forwarding an invalidation message from the head down the list, and the requester becomes the new head of the list. However, the distribution of individual directory entries increases the latency and complexity of the memory references. To reduce latency and to support additional functions, the SCI working committee has proposed some enhancements such as converting sharing lists to sharing trees, request combining, and support for queue-based locks.

COMA DSM SYSTEMS
The COMA architecture (see Figure C) uses local memories of the clusters as huge caches for data blocks from virtual shared address spaces (attraction memories). There is no physical memory home location predetermined for a particular data item, and it can be replicated and migrated in attraction mem-

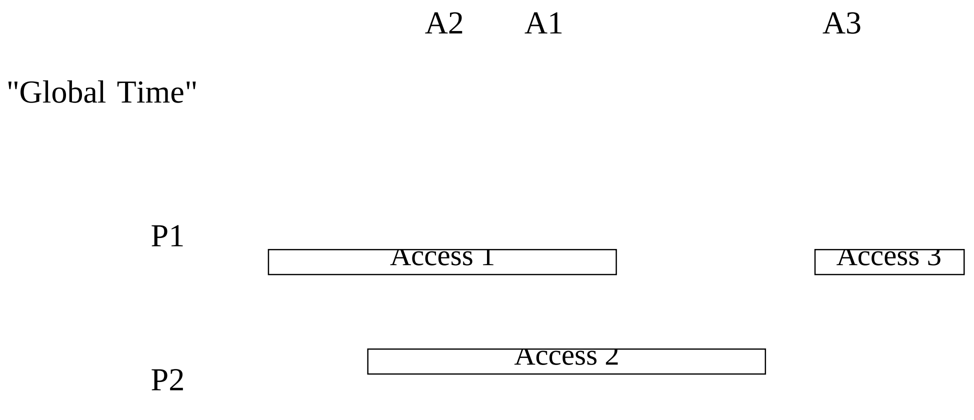
Sequential consistency

All processors see same (global) order and
order respects all internal orders (not nec. real time)

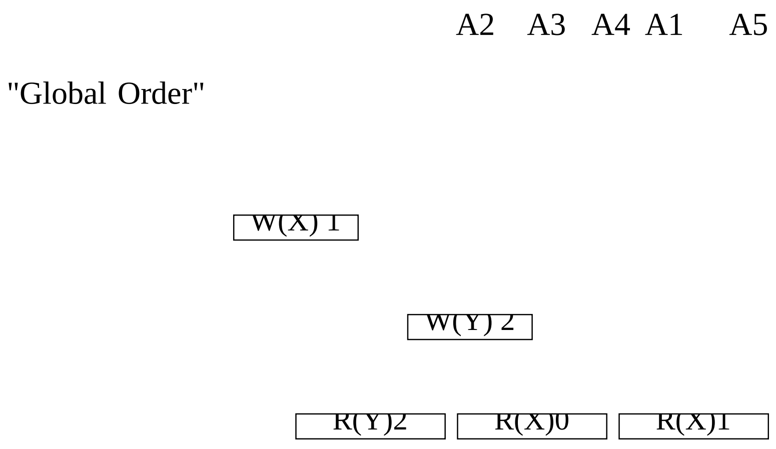
$P_1 : W(X)1$

$P_2 : \qquad\qquad W(Y)2$

$P_3 : \qquad\qquad\qquad\qquad R(Y)2 \; R(X)0 \;\; R(X)1$



Atomic Consistency − global total order respecting access intervals

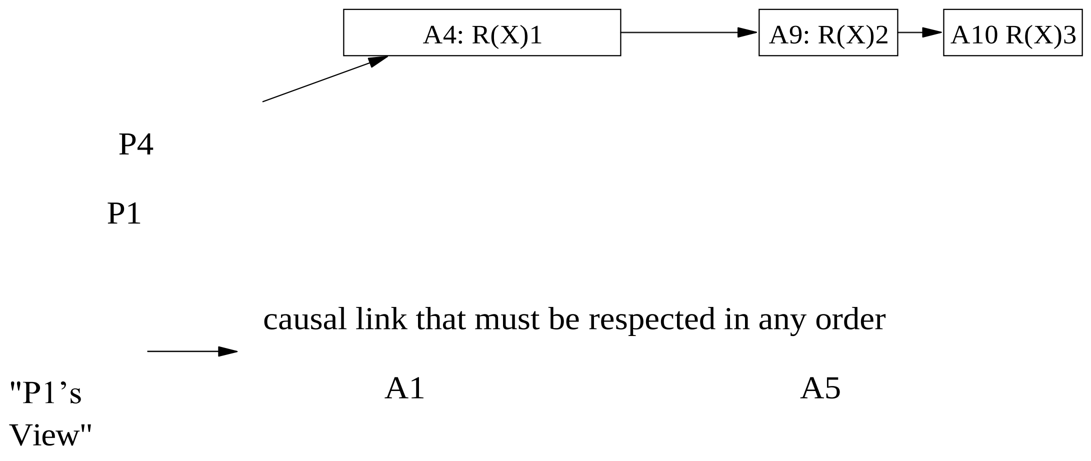


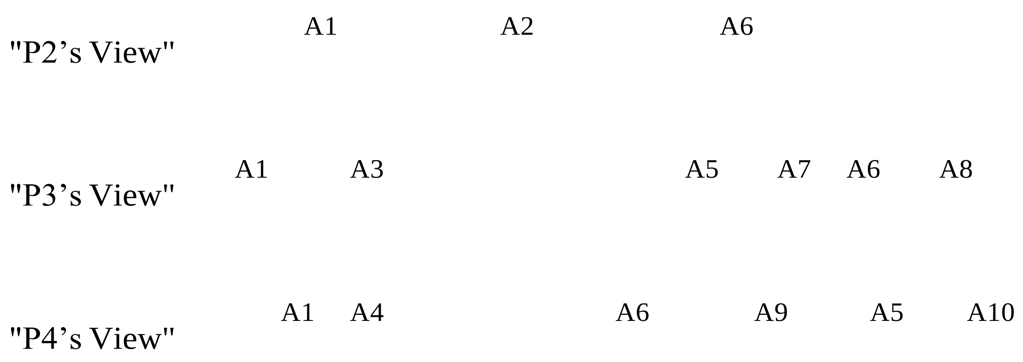
Sequential Consistency − global total order (not nec. respecting access intervals)

Causal consistency

Processors may see different order
all orders respect causal order (internal and r-w)

P_1 :	$W(X)1$		$W(X)3$		
P_2 :	$R(X)1$	$W(X)2$			
P_3 :	$R(X)1$			$R(X)3$	$R(X)2$
P_4 :	$R(X)1$			$R(X)2$	$R(X)3$





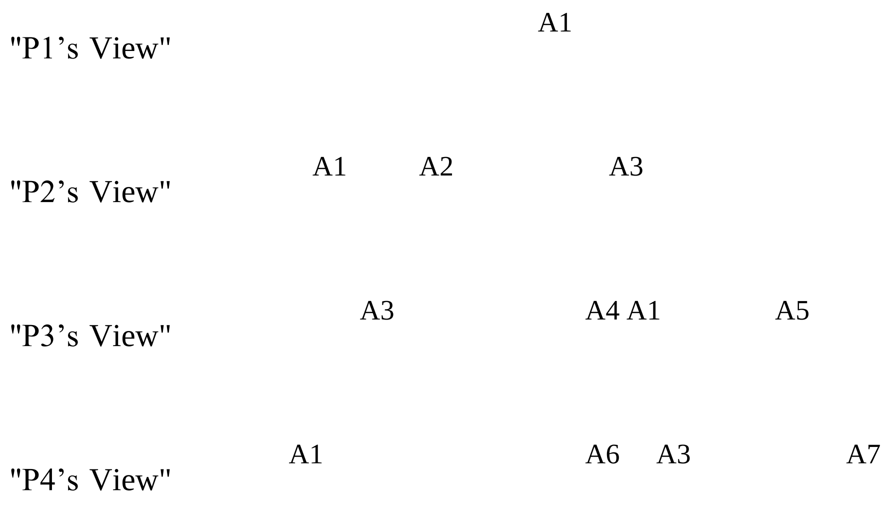
Causal Consistency − no global total order; causal partial order only Each processor's order respects internal order and Write−Read causality

4.3.1 Processor consistency

Writes from same processor are in order
Writes from different processors not constrained

$$\begin{array}{lllll}
P_1: & W(X)1 & & & \\
P_2: & & R(X)1 & W(X)2 & \\
P_3: & & & & R(X)1 \quad R(X)2 \\
P_4: & & & & R(X)2 \quad R(X)1
\end{array}$$

→ causal link that need not be respected → internal link that is respected



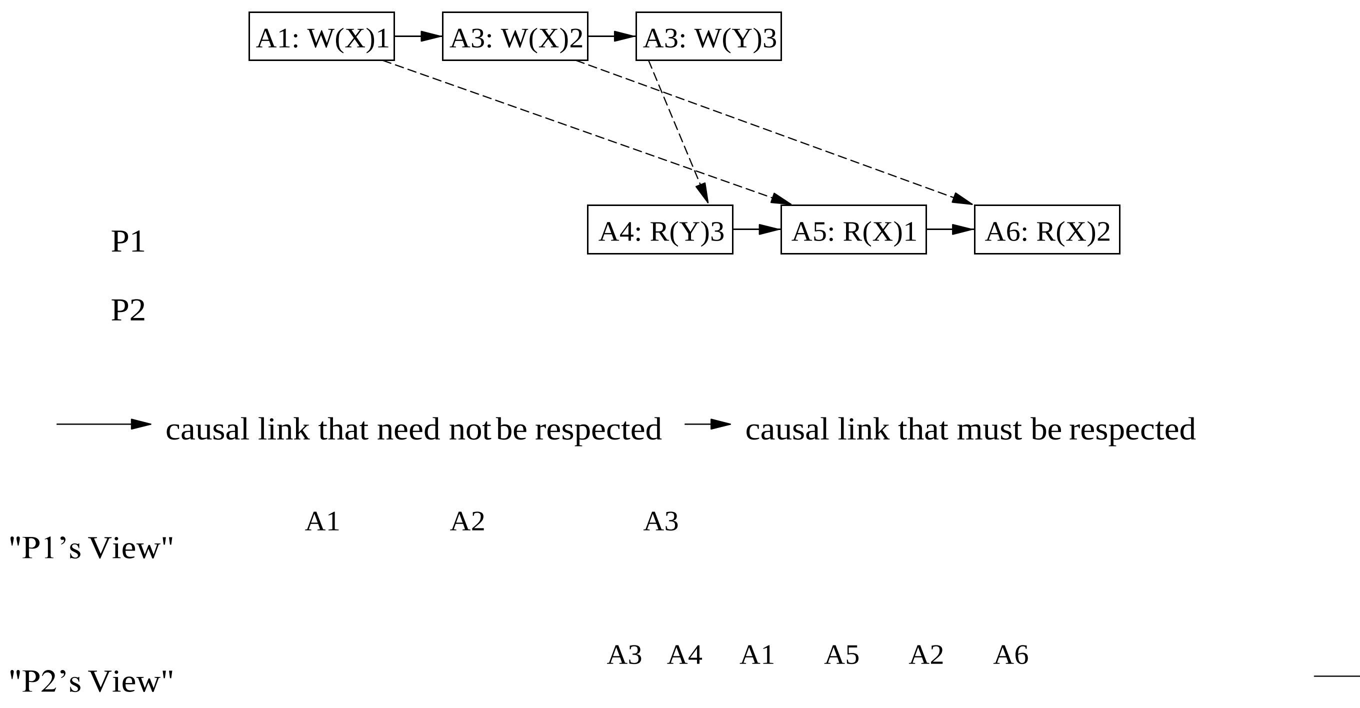
Processor Consistency − no global total order; partial order on writes by same processor Each processor's order respects internal order and order of writes by same processor

Slow memory consistency

Writes from same processor to same location are in order
Writes from different processors or locations not constrained

$P_1 : W(X)1 \; W(X)2 \; W(Y)3$

$P_2 : \qquad\qquad\qquad R(Y)3 \quad R(X)1 \;\; R(X)2$



Slow Memory Consistency − no global total order, no constraints across memory locations

Each processor's order respects its internal order and order of writes to same memory by same processor

4.3.2 Synchronization Access Consistency Models

Accesses to *synchronization variables* distinguished from accesses to ordinary shared variables

Weak consistency

- Accesses to *synchronization variables* are sequentially consistent
- No access to a synchronization variable is issued by a processor before all previous *read/write* data accesses have been performed
  (i.e., synch waits until all ongoing accesses complete)
- No *read/write* data access is issued by a processor before a previous access to a synchronization variable has been performed
  (i.e., all new accesses must wait until synch is performed)
- in effect, system "settles" at synch.

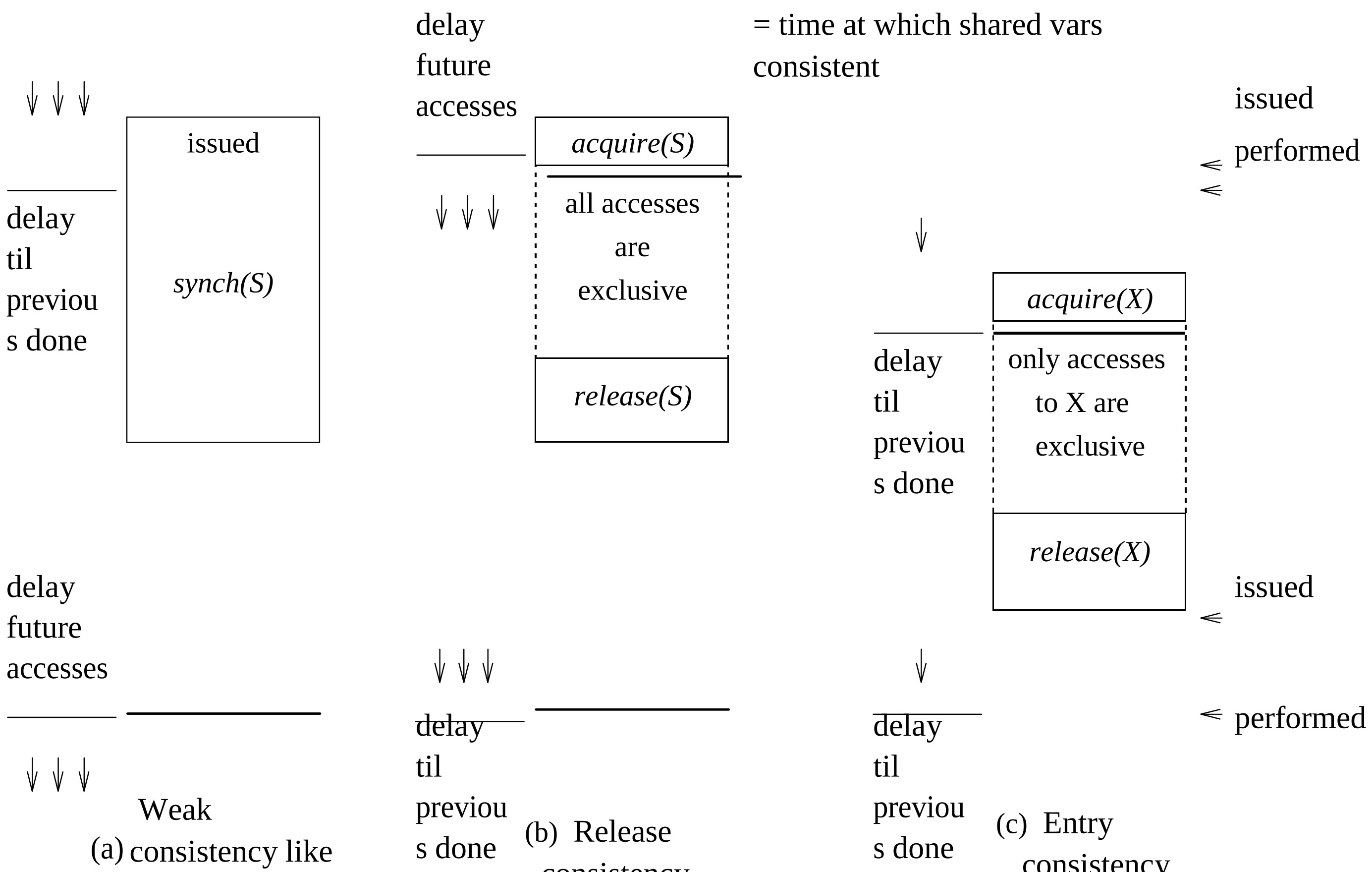
Release consistency

The synchronization access (*synch*(*S*)) in the weak consistency model can be refined as a pair of *acquire*(*S*) and *release*(*S*) accesses. Shared variables in the critical section are made consistent when the release operation is per- formed.
(i.e., *S* "locks" access to shared variables it protects, and release is not com- pleted until all accesses to them are also completed).

Entry consistency

acquire and *release* are applied to general variables.

Each variable has an implicit synchronization variable that may be acquired to prevent concurrent access to it.



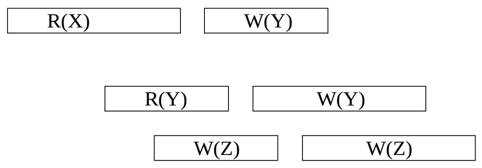
(a) Weak consistency like barrier sync

but local to process – only sync when necessary

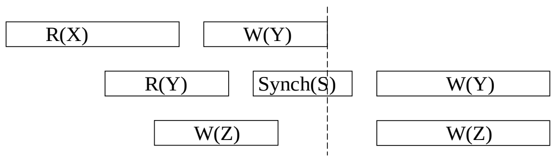
(b) Release consistency Processor consistency w.r.t. S

All vars in DSM system

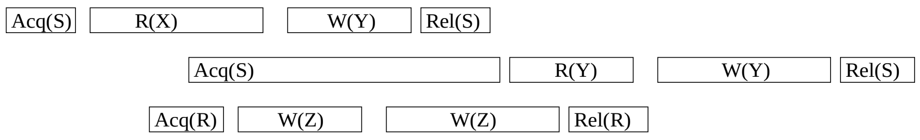
(c) Entry consistency Consistency w.r.t. memory object X across all procs



No synchronization

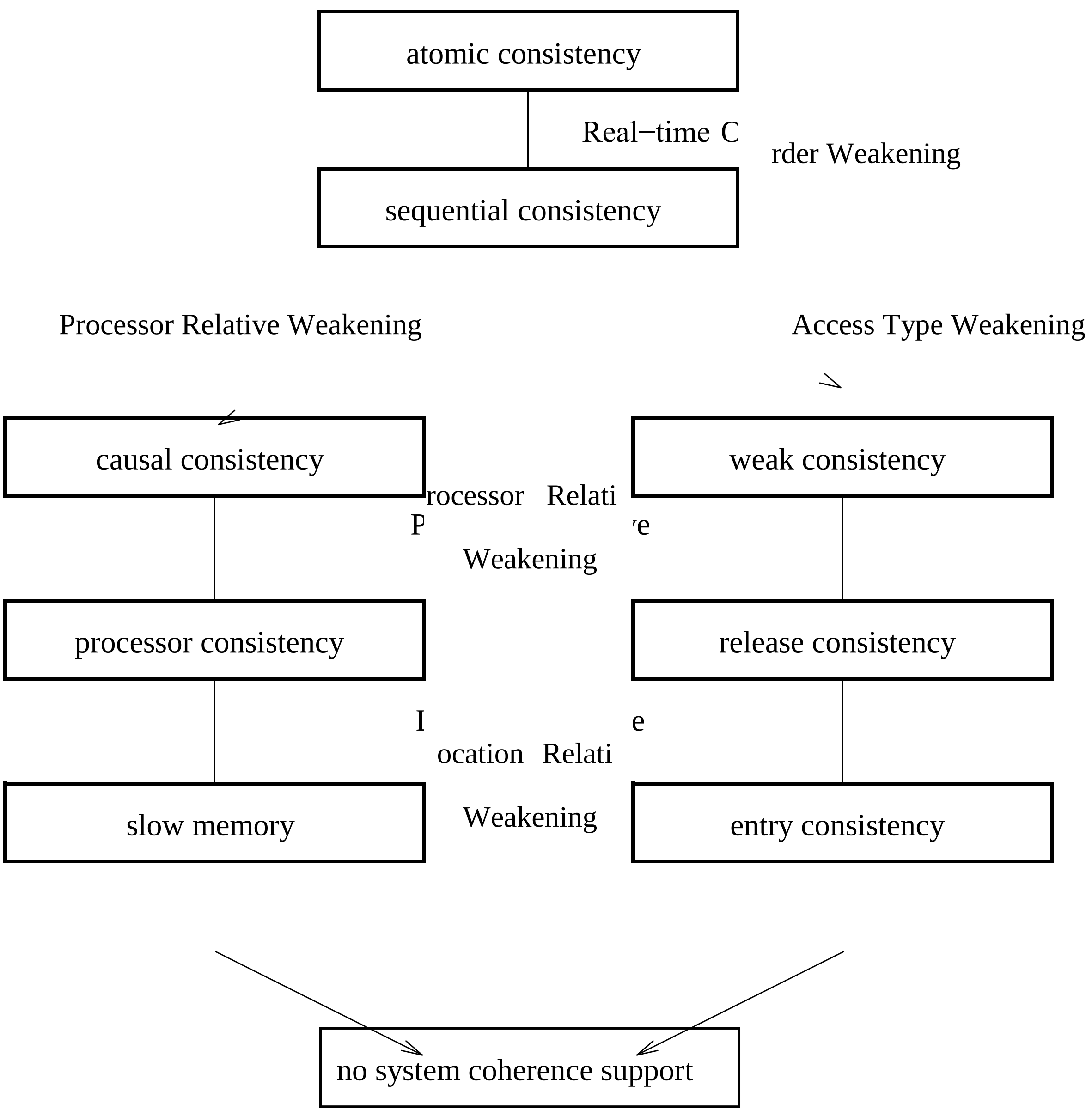


Weak consistency



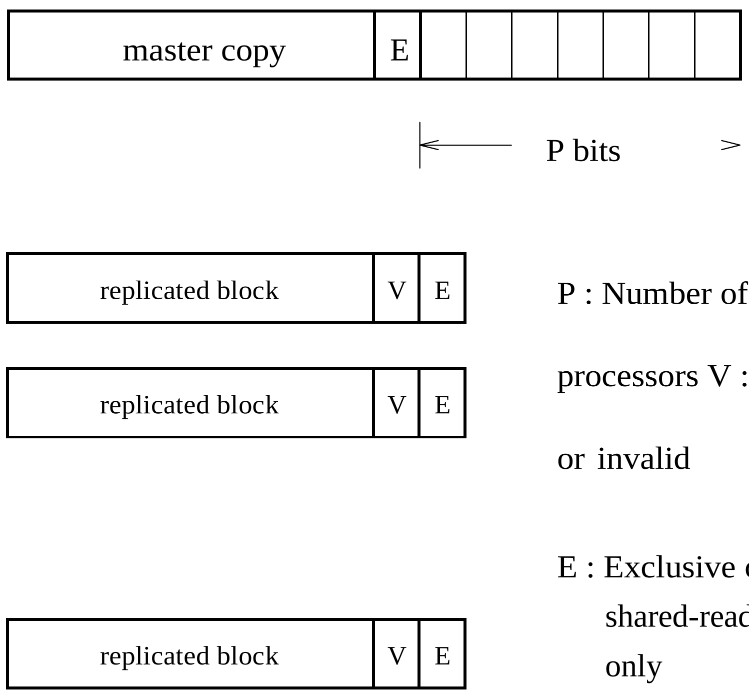
Release consistency

Taxonomy


atomic consistency
Real−time C rder Weakening
sequential consistency
Processor Relative Weakening
Access Type Weakening
causal consistency
weak consistency
P rocessor Relati ve
Weakening
processor consistency
release consistency
l ocation Relati e
Weakening
slow memory
entry consistency
no system coherence support

4.4 Multiprocessor Cache Systems

Cache directory



P : Number of

processors V : Valid

or invalid

E : Exclusive or
shared-read-
only

V bit for validity (in replicas), E bit for exclusive access (in all)
May also include *private* (= not shared) bit and/or *dirty* (= modified) bit.

Cache coherency protocols

write-invalidate and write-

update Write-invalidate

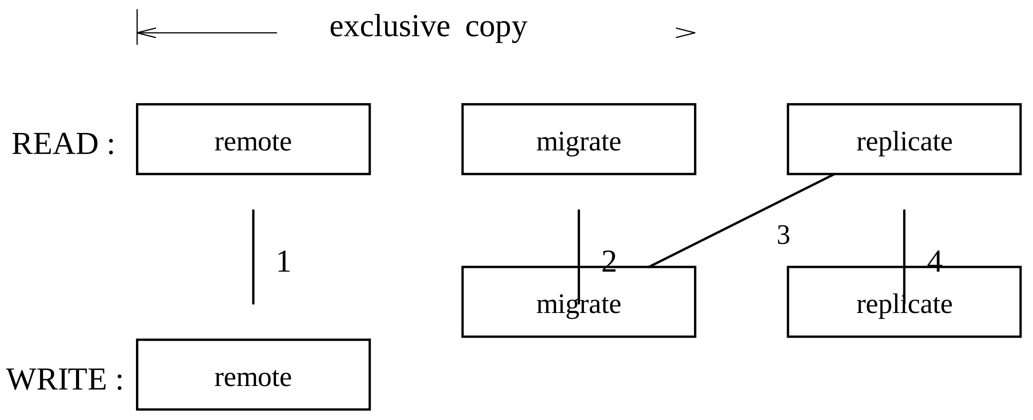
- Read hit
- Read miss: transfer block, set P-, V-, and E-bit.
- Write hit: invalidate cache copies, write and set E-bit
- Write miss: like read miss/write hit

Hardware mechanisms

- Directory-based
- Snooping cache

4.5 DSM implementation

Memory management algorithms



1 : Central server algorithm (SRSW)

2 : Migration algoritm (SRSW)

3 : Read-replication algorithm (MRSW)

4 : Full-replication algorithm (MRMW)

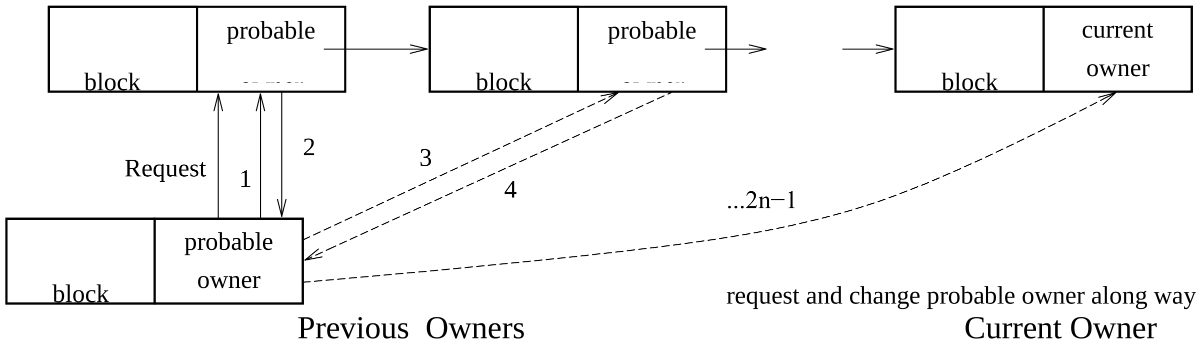
- Read-remote-write-remote: long network delay, trivial consistency
- Read-migrate-write-migrate: thrashing and false sharing
- Read-replicate-write-migrate: write-invalidate
- Read-replicate-write-replicate; full concurrency, atomic update

Considerations:

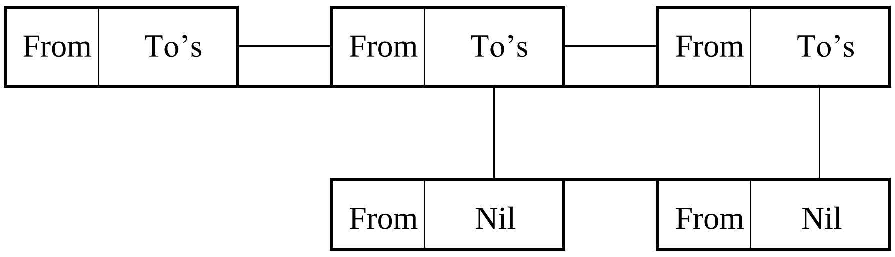
- Block granularity
- Block transfer communication overhead
- Read/write ratio
- Locality of reference
- Number of nodes and type of interaction

4.5.1 Distributed implementation of Directory

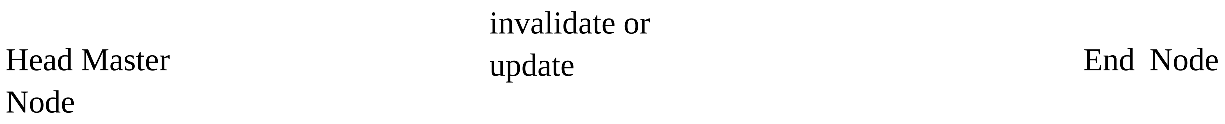
Locating Block Owner:

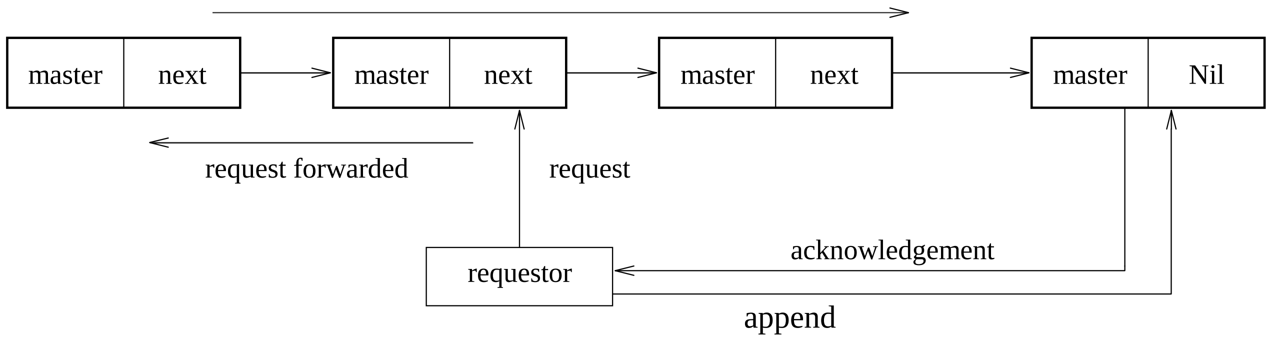


Maintaining Copy List:

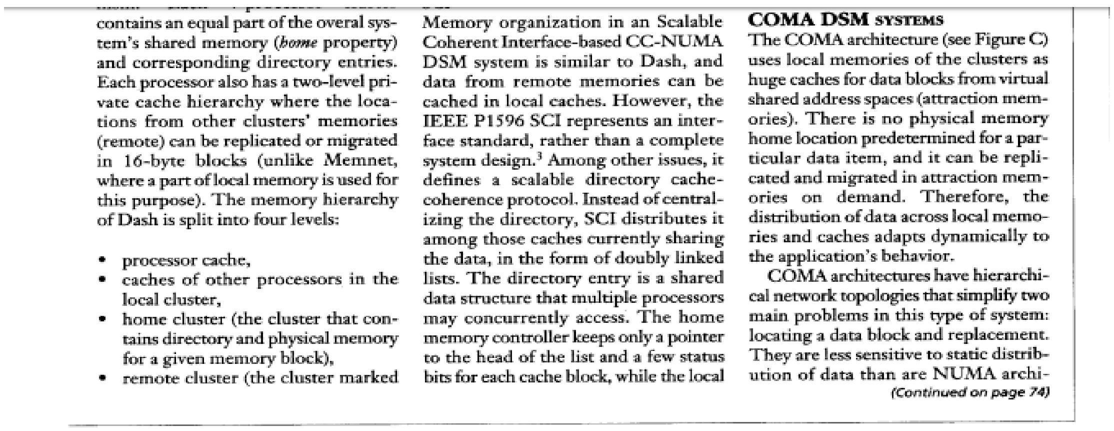


(a) Spanning tree representation of copy set





(b) Linked list representation of copy set



contains an equal part of the overal system's shared memory (*home* property) and corresponding directory entries. Each processor also has a two-level private cache hierarchy where the locations from other clusters' memories (remote) can be replicated or migrated in 16-byte blocks (unlike Memnet, where a part of local memory is used for this purpose). The memory hierarchy of Dash is split into four levels:

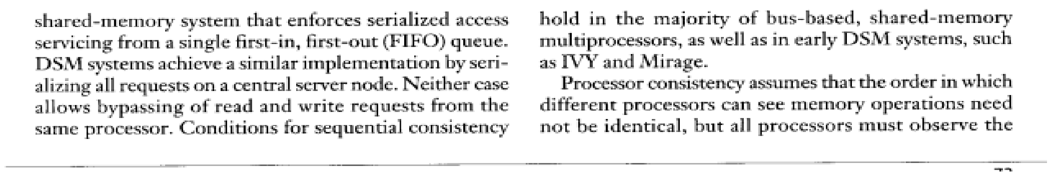
- processor cache,
- caches of other processors in the local cluster,
- home cluster (the cluster that contains directory and physical memory for a given memory block),
- remote cluster (the cluster marked

Memory organization in an Scalable Coherent Interface-based CC-NUMA DSM system is similar to Dash, and data from remote memories can be cached in local caches. However, the IEEE P1596 SCI represents an interface standard, rather than a complete system design.[3] Among other issues, it defines a scalable directory cache-coherence protocol. Instead of centralizing the directory, SCI distributes it among those caches currently sharing the data, in the form of doubly linked lists. The directory entry is a shared data structure that multiple processors may concurrently access. The home memory controller keeps only a pointer to the head of the list and a few status bits for each cache block, while the local

COMA DSM SYSTEMS
The COMA architecture (see Figure C) uses local memories of the clusters as huge caches for data blocks from virtual shared address spaces (attraction memories). There is no physical memory home location predetermined for a particular data item, and it can be replicated and migrated in attraction memories on demand. Therefore, the distribution of data across local memories and caches adapts dynamically to the application's behavior.

COMA architectures have hierarchical network topologies that simplify two main problems in this type of system: locating a data block and replacement. They are less sensitive to static distribution of data than are NUMA archi-

(Continued on page 74)



shared-memory system that enforces serialized access servicing from a single first-in, first-out (FIFO) queue. DSM systems achieve a similar implementation by serializing all requests on a central server node. Neither case allows bypassing of read and write requests from the same processor. Conditions for sequential consistency hold in the majority of bus-based, shared-memory multiprocessors, as well as in early DSM systems, such as IVY and Mirage.

Processor consistency assumes that the order in which different processors can see memory operations need not be identical, but all processors must observe the

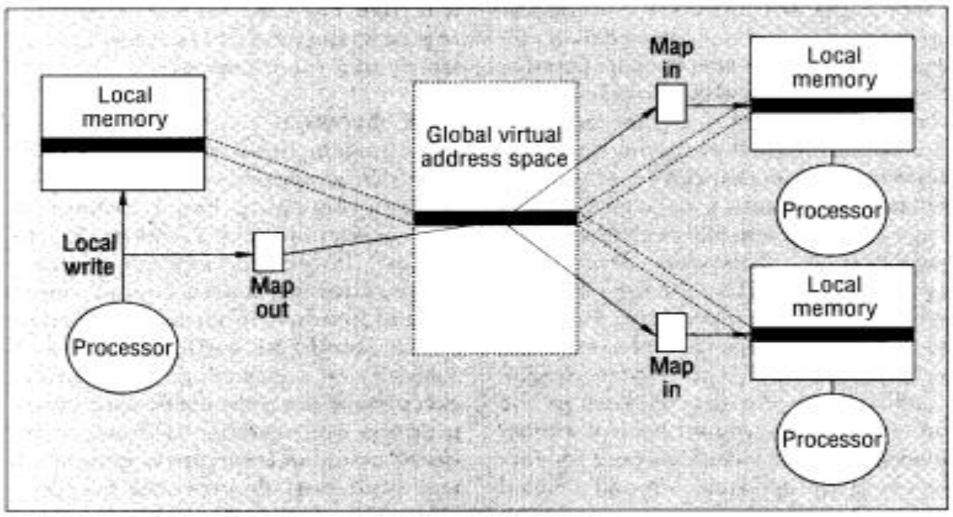
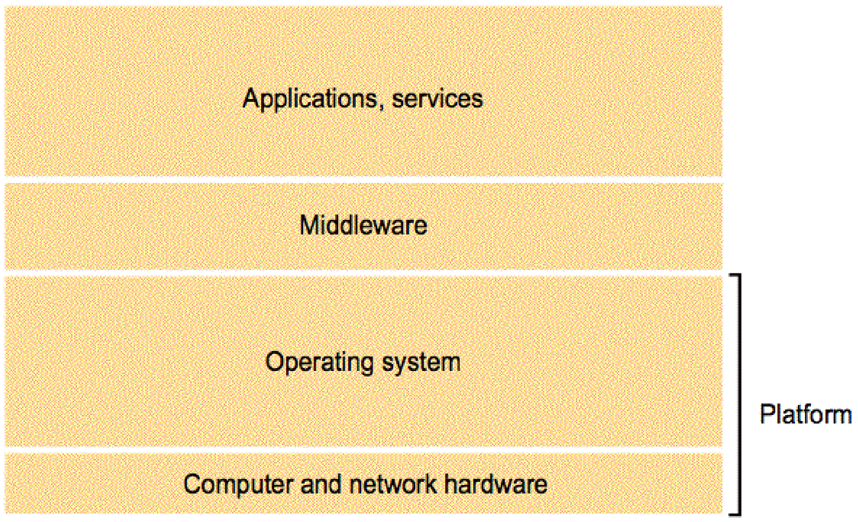


Figure D. Reflective memory DSM architecture.

4.6 Models of Distributed Computation

4.6.1 Software Layers

First, consider the software architecture of the components of a distributed system.



The lower two layers comprise the platform, such as Intel x86/Windows or PowerPC/MacOS X, that provides OS-level services to the upper layers.

The middleware sits between the platform and the application and its purpose is to mask heterogeneity and provide a consistent programming model for application developers. Some of the abstractions provided by middleware include the following:

- Remote method invocation
- Group communication
- Event notification
- Object replication
- Real-time data transmission

Examples of middleware include the following:

- Java RMI
- CORBA
- DCOM

Atop the middleware layer sits the application layer. The application layer provides application-specific functionality. Depending on the application, it may or may not make sense to take advantage of existing middleware.

4.6.2 System Architectures

The application layer defines the functional role of each component in a distributed system, and each component may have a different functional role. There are several common architectures employed by distributed systems. The choice of architecture can impact the design considerations described below:

- Responsiveness - how quickly does the system respond to requests?
- Throughput - how many requests can the system handle (per second, for example)?
- Load Distribution - are requests distributed evenly among components of the system?
- Fault Tolerance - can the system continue to handle requests in the face of a failed component?
- Security - does the system ensure that sensitive resources are guarded against attack?

Common architectures for distributed systems are as follows:

Client-Server

The client-server model is probably the most popular paradigm. The server is responsible for accepting, processing, and replying to requests. It is the producer. The client is purely the consumer. It requests the services of the server and accepts the results.

The basic web follows the client-server model. Your browser is the client. It requests web pages from a server (e.g., google.com), waits for results, and displays them for the user.

In some cases, a web server may also act as a client. For example, it may act as a client of DNS or may request other web pages.

Multiple Servers

In reality, a web site is rarely supported with only one server. Such an implementation would not be scalable or reliable. Instead, web sites such as Google or CNN are hosted on many (many many) machines. Services are either replicated, which means that each machine can perform the same task, or partitioned, which means that some machines perform one set of tasks and some machines perform another set of tasks. For example, a site like CNN might serve images from one set of machines and HTML from another set of machines.

Proxies and Caches

To reduce latency, load on the origin server, and bandwidth usage, proxies and caches are also used to deliver content. An end host (your browser) may cache content. In this case, when you first request content, your browser stores a copy on your local machine. Subsequent requests for the same content can be fulfilled by using the cache rather than requesting the content from the origin server.

An organization, like USF, may also deploy a proxy server that can cache content and deliver it to any client within the organization. Again, this reduces latency, and it also reduces bandwidth usage. Suppose that several hundred USF students download the same YouTube video. If a proxy server caches the video after the first student's request, subsequent requests can be satisfied by using the cached content, thereby reducing the number of external requests by several hundred.

CDNs, like Akamai, also fall into this category. However, CDNs work a bit differently than traditional proxy servers. CDNs actively replicate content throughout the network in a *push*-based fashion. When a customer (e.g., CNN) updates its content, the new content is replicated throughout the network. In contrast, a proxy server will cache new content when it is requested by the first client.

4.6.3 P2P

The peer-to-peer model assumes that each entity in the network has equivalent functionality. In essence, it can play the role of a client or a server. Ideally, this reduces bottlenecks and enables each entity to contribute resources to the system. Unfortunately, it doesn't always work that way. One of the early papers on peer-to-peer systems was Free Riding on Gnutella, a paper that demonstrated that peers often free ride by taking resources (downloading files, in this case) and never contributing resources (uploading files).

In addition, enabling communication in such a system is challenging. First, peers must locate other peers in order to participate in the system. This is rarely done in a truly distributed or peer-to-peer fashion. For example, Napster, often cited (controversially) as the first real example of peer-to-peer computing, used a centralized mechanism for joining the network and searching for content. Searching for content or other resources is the second big challenging in implementing peer-to-peer systems. It can be very inefficient to locate resources in a peer-to-peer system and a hybrid, or partially centralized, solution is often employed.

Hierarchical or superpeer systems, like Skype, are also widely used. In these systems, peers are organized in a tree-like structure. Typically, more capable peers are elected to become superpeers (or supernodes). Superpeers act on behalf of downstream peers and can reduce communication overhead.

Mobile Code/Agents

The previous models assume that the client/server/peer entities exchange data. The mobile code model assumes that components may exchange code. An example of this is Java Applets. When your browser downloads and applet, it downloads some Java code that it then runs locally. The big issue with this model is that it introduces security risks. No less a security threat are mobile agents -- processes that can move from machine to machine.

Network Computers/Thin Clients

The network computer model assumes that the end user machine is a low-end computer that maintains a minimal OS. When it boots, it retrieves the OS and files/applications from a central server and runs applications locally. The thin client model is similar, though assumes that the process runs remotely and the client machine simply displays results (e.g., X-windows and VNC).

This model has been around for quite some time, but has recently received much attention. Google and Amazon are both promoting "cloud computing". Sun's Sun Ray technology also makes for an interesting demonstration. Though this model has yet to see success, it is beginning to look more promising.

Mobile Devices

There is an increasing need to develop distributed systems that can run atop devices such as cell phones, cameras, and MP3 players. Unlike traditional distributed computing entities, which communicate over the Internet or standard local area networks, these devices often communicate via wireless technologies such as Bluetooth or other low bandwidth and/or short range mechanisms. As a result, the geographic location of the devices impacts system design. Moreover, mobile systems must take care to consider the battery constraints of the participating devices. System design for mobile ad hoc networks (MANETs), sensor networks, and delay/disruption tolerant networks (DTNs) is a very active area of research.

4.6.4 Fundamental Models

Or, understanding the characteristics that impact distributed system performance and operation.

Interaction

Fundamentally, distributed systems are comprised of entities that communicate and coordinate by passing messages. The following characteristics of communication channels impact the performance of the system:

- Latency - the time between the sending of a message at the source and the receipt of the message at the destination.
- Bandwidth - the total amount of information that can be transmitted over a given time period (e.g., Mbits/second).
- Jitter - "the variation int he time taken to deliver a series of messages." (Coulouris et al)

Additionally, coordination of the actions of entities in a distributed system is impacted by the fact that each entity will have a different *clock drift rate*. Synchronous distributed systems that rely on certain actions happening at the same time can only be built if you can guarantee bounds on system resources and clock drift rates. Most of the systems that we will discuss are asynchronous; there are no guarantees about the time at which actions will occur.

Generally, it is sufficient to know the order in which events occur. A logical clock is a counter that allows a system to keep track of when events occur *in relation to other events*.

Failure

It is important to understand the kinds of failures that may occur in a system.

- Failstop: A process halts and remains halted. Other processes can detect that the process has failed.
- Crash: A process halts and remains halted. Other processes may not be able to detect this state.
- Omission: A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
- Send-omission: A process completes a send, but the message is not put in its outgoing message buffer.
- Receive-omission: A message is put in a process's incoming message buffer, but that process does not receive it.
- Arbitrary (Byzantine): Process/channel exhibits arbitrary behavior: it may send/transmit arbitrary messages at arbitrary times, commit missions; a process may stop or take an incorrect step.
- Timing failure: Clock drift exceeds allowable bounds.

Security

There are several potential threats a system designer need be aware of:

- Threats to processes - An attacker may send a request or response using a false identity (spoofing).
- Threats to communication channels - An attacker may eavesdrop (listen to messages) or inject new messages into a communication channel. An attacker can also save messages and replay them later.
- Denial of service - An attacker may overload a server by making excessive requests.

Cryptography and authentication are often used to provide security. Communication entities can use a shared secret (key) to ensure that they are communicating with one another and to encrypt their messages so that they cannot be read by attackers.

4.7 Failures in a Distributed System

4.7.1Types of Failures in Distributed Systems

There are different types of failure across the distributed system and few of them are given in this section as below

Crash failures: Crash failures are caused across the server of a typical distributed system and if these failures are occurred operations of the server are halt for some time. Operating system failures are the best examples for this case and the corresponding fault tolerant systems are developed with respect to these affects.

Timing failures: Timing failures are caused across the server of a distributed system. The usual behavior of these timing failures would be like that the server response time towards the client requests would be more than the expected range. Control flow out of the responses may be caused due to these timing failures and the corresponding clients may give up as they can't wait for the required response from the server and thus the server operations are failed due to this.

Omission failures: Omission failures are caused across the server due to lack or reply or response from the server across the distributed systems. There are different issues raised due to these omission failures and the key among them are server not listening or a typical buffer overflow errors across the servers of the distributed systems.

Byzantine failures: Byzantine failures are also know as arbitrary failures and these failures are caused across the server of the distributed systems. These failures cause the server to behave arbitrary in nature and the server responds in an arbitrary passion at arbitrary times across the distributed systems. Output from the server would be inappropriate and there could be chances of the malicious events and duplicate messages from the server side and the clients get arbitrary and unwanted duplicate updates from the server due to these failures.

4.8 Mutual exclusion in distributed system

4.8.1 Introduction

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.
Mutual exclusion in single computer system Vs. distributed system:
In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.

Requirements of Mutual exclusion Algorithm:

- No Deadlock:
  Two or more site should not endlessly wait for any message that will never arrive.
- No Starvation:
  Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section
- Fairness:
  Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.
- Fault Tolerance:
  In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

4.8.2 Solution to distributed mutual exclusion:
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
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

4.9 Election algorithm and distributed processing

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.

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

4.9.1 **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.

1. If coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
2. Now process P sends election message to every process with high priority number.
3. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.
4. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
5. 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 responds within time interval T' then it is assumed to have failed and algorithm is restarted.

2. **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 –

1. 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.
2. If process P2 receives message elect from processes on left, it responds in 3 ways:

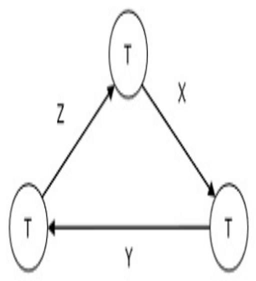
(I) If message received does not contain 1 in active list then P1 adds 2 to its active list and forwards the message.

3. (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.
4. (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.

Distributed Deadlock handling

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

For example, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.



Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely −

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

All of the three approaches can be incorporated in both a centralized and a distributed database system.

Deadlock Prevention

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any lock.

Deadlock Avoidance

The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose, namely **wait-die** and **wound-wait**. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows −

- **Wait-Die** − If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.
- **Wound-Wait** − If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

Deadlock Detection and Removal

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-forgraph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are −

- Choose the youngest transaction.
- Choose the transaction with fewest data items.
- Choose the transaction that has performed least number of updates.
- Choose the transaction having least restart overhead.
- Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

- One server may be selected as the center of control.
- The center of control may travel from one server to another.
- The responsibility of controlling may be shared by a number of servers.

Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

- Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
- In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
- If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur −

- Conflict between two transactions in the same site.
- Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows −

- **Distributed Wound-Die**
    - If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
    - If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.
- **Distributed Wait-Wait**
    - If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
    - If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

Distributed Deadlock Detection

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

- **Centralized Deadlock Detector** − One site is designated as the central deadlock detector.
- **Hierarchical Deadlock Detector** − A number of deadlock detectors are arranged in hierarchy.

- **Distributed Deadlock Detector** − All the sites participate in detecting deadlocks and removing them.

Distributed termination detection

A fundamental problem:

1. To determine if a distributed computation has terminated.
2. A non-trivial task since no process has complete knowledge of the global state, and global time does not exist.
3. A distributed computation is globally terminated if every process is locally terminated and there is no message in transit between any processes.
4. "Locally terminated" state is a state in which a process has finished its computation and will not restart any action unless it receives a message.
5. In the termination detection problem, a particular process (or all of the processes) must infer when the underlying computation has terminated.
   - A termination detection algorithm is used for this purpose.
   - Messages used in the underlying computation are called basic messages, and messages used for the purpose of termination detection are called control messages.
   - A termination detection (TD) algorithm must ensure the following:
     1 Execution of a TD algorithm cannot indefinitely delay the underlying computation.
     2 The termination detection algorithm must not require addition of new communication channels between processes.

Definition of Termination Detection

1. Let pi(t) denote the state (active or idle) of process pi at instant t.
2. Let ci,j(t) denote the number of messages in transit in the channel at instant t from process pi to process pj .
3. A distributed computation is said to be terminated at time instant t0 iff:
   (∀i:: pi(t0) = idle) ∧ (∀i, j:: ci,j(t0)=0).
   Thus, a distributed computation has terminated
   iff all processes have become idle and there is no message in transit in any channel.

Termination detection Using Distributed Snapshots

- The algorithm assumes that there is a logical bidirectional communication channel between every pair of processes.
- Communication channels are reliable but non-FIFO. Message delay is arbitrary but finite.

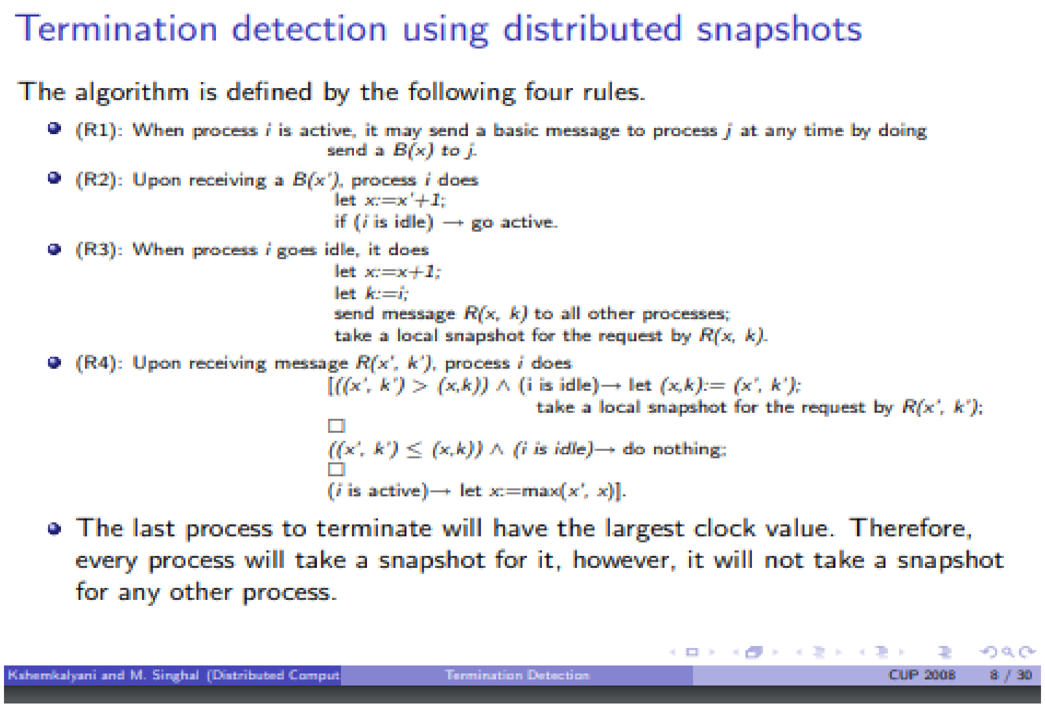
Main idea:

- When a process goes from active to idle, it issues a request to all other processes to take a local snapshot, and also requests itself to take a local snapshot.
- When a process receives the request, if it agrees that the requester became idle before itself, it grants the request by taking a local snapshot for the request.
- A request is *successful* if all processes have taken a local snapshot for it.
- The requester or any external agent may collect all the local snapshots of a request.
- If a request is successful, a global snapshot of the request can thus be obtained and the recorded state will indicate termination of the computation,

Termination detection using distributed snapshots

Formal Description

- Each process *i* maintains a *logical clock* denoted by x, initialized to zero at the start of the computation.
- A process increments its x by one each time it becomes idle.
- A basic message sent by a process at its logical time x is of the form $B(x)$.
- A control message that requests processes to take local snapshot issued by process i at its logical time x is of the form $R(x, i)$.
- Each process synchronizes its logical clock x loosely with the logical clocks x's on other processes in such a way that it is the maximum of clock values ever received or sent in messages.
- A process also maintains a variable k such that when the process is idle, (x,k) is the maximum of the values (x, k) on all messages $R(x, k)$ ever received or sent by the process.
- Logical time is compared as follows: $(x, k) > (x', k')$ iff $(x > x')$ or $((x{=}x')$ *and* $(k{>}k'))$, i.e., a tie between x and x' is broken by the process identification numbers k and k'.



Termination detection using distributed snapshots

The algorithm is defined by the following four rules.

- (R1): When process *i* is active, it may send a basic message to process *j* at any time by doing

```
send a B(x) to j.
```

- (R2): Upon receiving a $B(x')$, process *i* does

```
let x:=x'+1;
if (i is idle) → go active.
```

- (R3): When process *i* goes idle, it does

```
let x:=x+1;
let k:=i;
send message R(x, k) to all other processes;
take a local snapshot for the request by R(x, k).
```

- (R4): Upon receiving message $R(x', k')$, process *i* does

```
[((x', k') > (x,k)) ∧ (i is idle)→ let (x,k):= (x', k');
                               take a local snapshot for the request by R(x', k');
□
((x', k') ≤ (x,k)) ∧ (i is idle)→ do nothing;
□
(i is active)→ let x:=max(x', x)].
```

- The last process to terminate will have the largest clock value. Therefore, every process will take a snapshot for it, however, it will not take a snapshot for any other process.

Kshemkalyani and M. Singhal (Distributed Comput Termination Detection CUP 2008 8 / 30