

# Client-Centric Consistency Models

Lecture-26

# Introduction

- **Eventual consistency:**

- There is a special class of **distributed data stores** that characterized by the lack of simultaneous updates, most of the operations involve reading data. T
- hese data stores offers a **very weak consistency model**, called **eventual consistency**.

- **Client-centric consistency models:**

- Eventual consistent data stores work fine as long as clients always access the same replica.
- However, problems arise when different replicas are accessed over a short period of time.
- Assume the user performs several update operations and then disconnects again.
- Later, he accesses the database again, possibly after moving to a different location or by using a different access device.
- At that point, the user may be connected to a different replica than before.
- If the updates performed previously have not yet been propagated, the user will notice inconsistent behavior.
- In particular, he would expect to see all previously made changes, but instead, it appears as if nothing at all has happened.

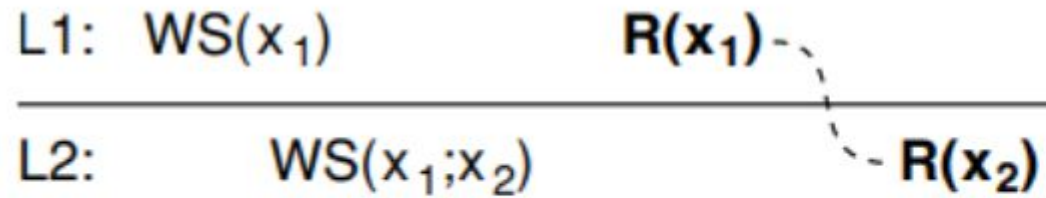
# Cont..

- By introducing special **client-centric consistency models**, the previous problem can be alleviated in a relatively cheap way.
- In essence, client-centric consistency provides guarantees for a single client.
- To explain these models, we consider a data store that is physically distributed across multiple machines.
- When a process accesses the data store, it generally connects to the nearest available copy, although any copy will be possible.
- All read and write operations are performed on that local copy.
- Updates are eventually propagated to the other copies.
- To simplify matters, we assume that data items have an associated owner, which is the only process that is permitted to modify that item.
- In this way, we avoid write-write conflicts.

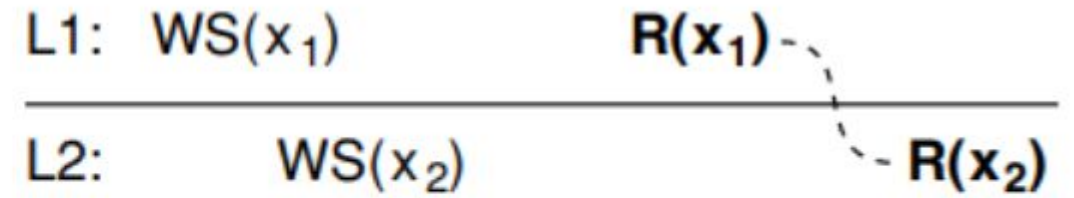
# Cont..

- Let  $x_i[t]$  denote the version of data item  $x$  at local copy  $L_i$  at time  $t$ .
- $x_i[t]$  is the result of a series of write operations denoted as  $WS(x_i[t])$  at  $L_i$  that took place since initialization until time  $t$ .
- If the same operations have been replicated at local copy  $L_j$  at a later time  $t_2$ , we write  $WS(x_i[t_1]; x_j[t_2])$ .
- **Monotonic Reads:**
  - In a monotonic-read consistent data store, if a process reads the value of a data item  $x$ , any successive read operation on  $x$  by that process will always return that same value or a more recent value.
  - **Example:** Consider a distributed email database, when a user open mailbox in San Francisco and notice a new email is present. When the user later flies to New York and opens his mailbox again, monotonic-read consistency guarantees that the messages that were in the mailbox in San Francisco will also be in the mailbox when it is opened in New York.

## Cont..



(a)

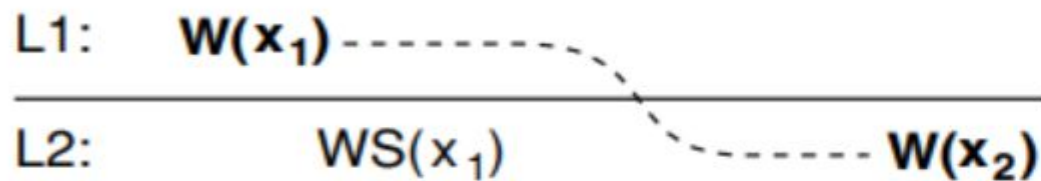


(b)

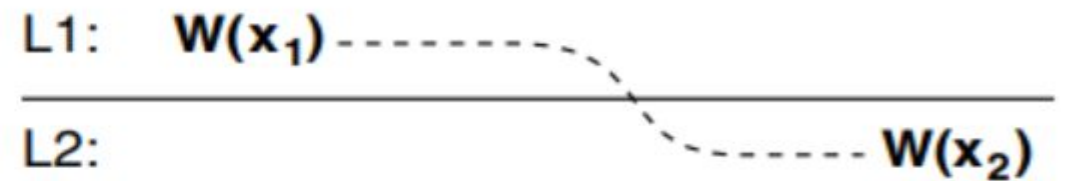
- In the figure (a), process P first performs a read operation on  $x$  at  $L_1$  shown as  $R(x_1)$  returning the value of  $x_1$ . This value results from the write operations  $WS(x_1)$  performed at  $L_1$
- Later, P performs a read operation on  $x$  at  $L_2$ , shown as  $R(x_2)$ .
- To guarantee monotonic-read consistency, all operations in  $W(x_1)$  should have been propagated to  $L_2$  before the second read operation takes place.
- In other words, we need to know for sure that  $WS(x_1)$  is part of  $WS(x_2)$  which is expressed as  $WS(x_1; x_2)$ .

## Cont..

- In the figure (b), shows a situation in which monotonic-read consistency is not guaranteed.
- After process P has read  $x_1$  at  $L_1$  it later performs the operation  $R(x_2)$  at  $L_2$ .
- However, only the write operations  $WS(x_2)$  have been performed at  $L_2$ .
- No guarantees are given that this set also contains all operations contained in  $WS(x_1)$ .
- **Monotonic Writes:**
  - In a monotonic-write consistent data store, a write operation by a process on a data item  $x$  is completed before any successive write operation on  $x$  by the same process.
  - In other words, no matter where that operation was initialized, for each node the new write of same process must wait for old write propagate to local copy.



(a)



(b)

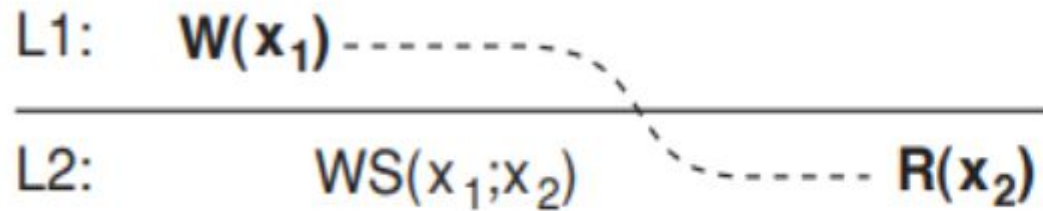
# Cont..

- In the figure (a), a process  $P$  performs a write operation on  $x$  at local copy  $L_1$ .
  - Later,  $P$  performs another write operation on  $x$ , but this time at  $L_2$ .
  - To ensure monotonic-write consistency, it is necessary that the previous write operation at  $L_1$  has already been propagated to  $L_2$ .
  - This is why we have  $W(x_1)$  at  $L_2$  and before  $W(x_2)$ .
- 
- In the figure (b), in which monotonic-write consistency is not guaranteed.
  - What is missing is the propagation of  $W(x_1)$  to copy  $L_2$ .
  - By the definition of monotonic-write consistency, write operations by the same process are performed in the same order as they are initialized.

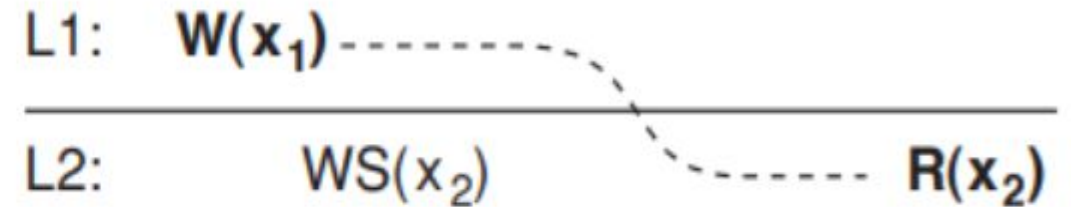
# Cont..

- **Read Your Writes:**

- In a read-your-writes consistent data store, the effect of a write operation by a process on data item  $x$  will always be seen by a successive read operation on  $x$  by the same process.
- In other words, a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.



(a)



(b)



# Cont..

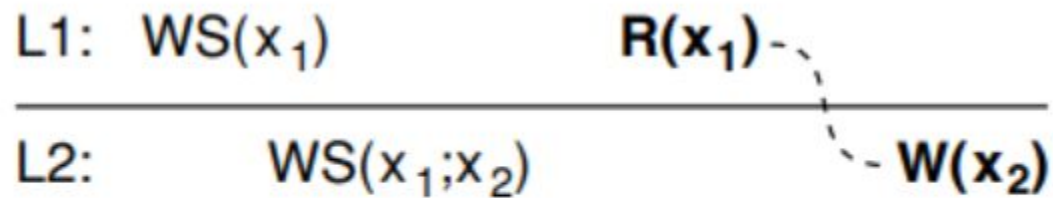
- In the figure (a), process P performed a write operation  $W(x_1)$  and later a read operation at a different local copy.
- Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation.
- This is expressed by  $W(x_1; x_2)$ , which states that  $W(x_1)$  is part of  $WS(x_2)$ .
- In the figure (b)  $W(x_1)$  has been left out of  $WS(x_2)$ , meaning that the effects of the previous write operation by process P have not been propagated to L2.

- **Writes Follow Reads:**

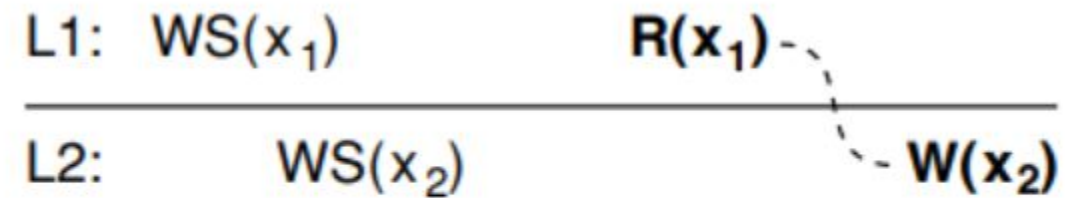
- In a writes-follow-reads consistent data store, a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.
- In other words, any successive write operation by a process on a data item will be performed on a copy of x that is up to date with the value most recently read by that process.

## Cont..

- Writes-follow-reads consistency can be used to guarantee that user can only post a reaction to an article only after they have seen the original article.




(a)



(b)

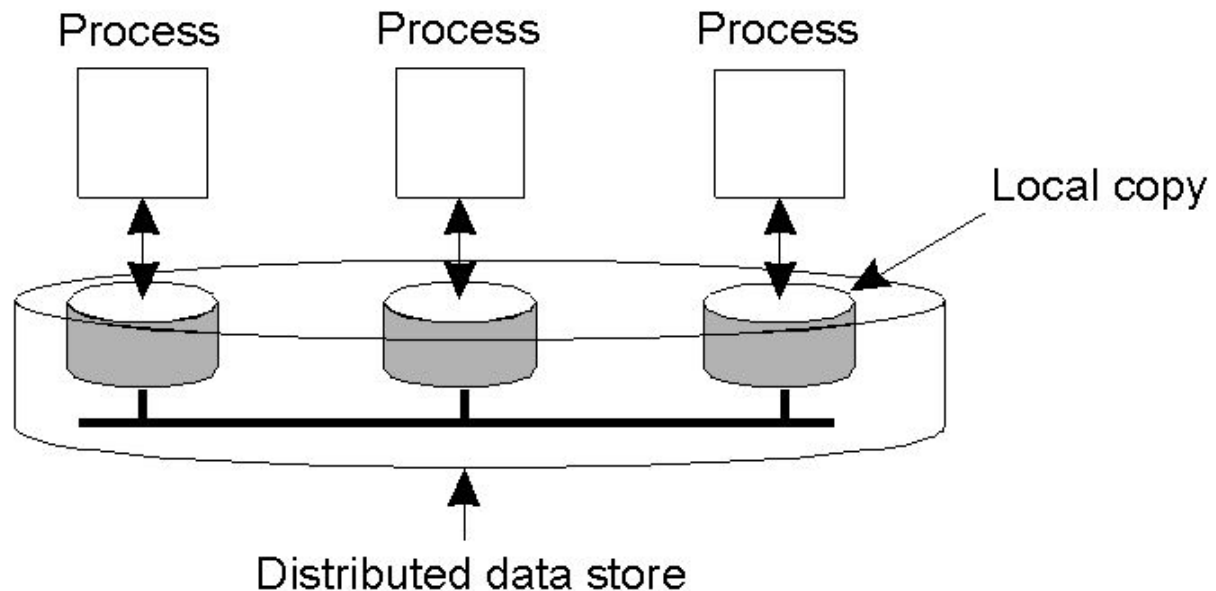
- In the figure (a), a process reads  $x$  at local copy  $L_1$  and the value is also propagates to  $L_2$ , where the same process later performs a write operation.
- In the figure (b), no guarantees are given that the operation performed at  $L_2$ , because the write operation is performed on a copy that doesn't receive the update that is on the  $L_1$ .

# Data-Centric Consistency Models



# Data-Centric Consistency Models

- A data-store can be read from or written to by any process in a distributed system.
- A local copy of the data-store (replica) can support “fast reads”.
- However, a **write** to a local replica needs to **be propagated to *all* remote replicas**.



- Various consistency models help to understand the various mechanisms used to achieve and enable this.

# What is a Consistency Model?

---

- A “consistency model” is a CONTRACT between a DS data-store and its processes.
- If the processes agree to the rules, the data-store will perform properly and as advertised.
- We start with *Strict Consistency*, which is defined as:
  - *Any read on a data item ‘x’ returns a value corresponding to the result of the most recent write on ‘x’ (regardless of where the write occurred).*

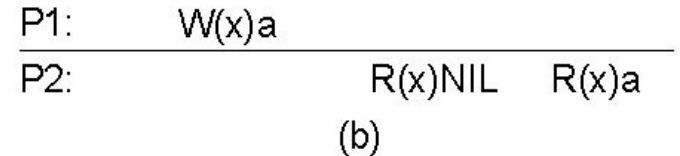
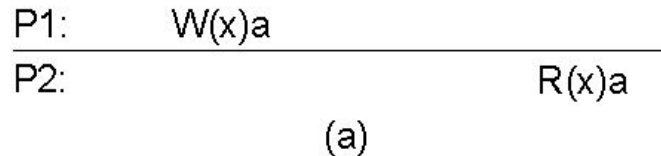
# Consistency Model Diagram Notation

---

- $W_i(x)a$  – a write by process 'i' to item 'x' with a value of 'a'. That is, 'x' is set to 'a'.
- (Note: The process is often shown as  $P_i$ ).
- $R_i(x)b$  – a read by process 'i' from item 'x' producing the value 'b'. That is, reading 'x' returns 'b'.
- Time moves from left to right in all diagrams.

# Strict Consistency Diagrams

---



- Behavior of two processes, operating on the same data item:
  - a) A strictly consistent data-store.
  - b) A data-store that is not strictly consistent.
- With *Strict Consistency*, all writes are *instantaneously visible* to all processes and *absolute global time order* is maintained throughout the DS. This is the consistency model “Holy Grail” – not at all easy in the real world, and all but *impossible* within a DS.
- So, other, less strict (or “weaker”) models have been developed ...

# Sequential Consistency

---

- A weaker consistency model, which represents a relaxation of the rules.
- It is also must easier (possible) to implement.
- Definition of “Sequential Consistency”:
  - *The result of any execution is the same as if the (read and write) operations by all processes on the data-store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*



# Sequential Consistency Diagrams

---

In other words: all processes see the same interleaving set of operations, regardless of what that interleaving is.

P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)b	R(x)a

(a)

P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

- a) A sequentially consistent data-store – the “first” write occurred *after* the “second” on all replicas.
- b) A data-store that is not sequentially consistent – it appears the writes have occurred in a non-sequential order, and this is **NOT** allowed.

# Problem with Sequential Consistency

---

- With this consistency model, adjusting the protocol to favour reads over writes (or vice-versa) can have a **devastating impact** on performance (refer to the textbook for the gory details).
- For this reason, other weaker consistency models have been proposed and developed.
- Again, a relaxation of the rules allows for these weaker models to make sense.

# Linearizability and Sequential Consistency (1)

---

Process P1	Process P2	Process P3
<pre>x = 1; print ( y, z);</pre>	<pre>y = 1; print (x, z);</pre>	<pre>z = 1; print (x, y);</pre>

- Three concurrently executing processes.

# Linearizability and Sequential Consistency (2)

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

```
x = 1;  
print ((y, z);  
y = 1;  
print (x, z);  
z = 1;  
print (x, y);
```

Prints: 001011

Signature:  
001011

(a)

```
x = 1;  
y = 1;  
print (x,z);  
print(y, z);  
z = 1;  
print (x, y);
```

Prints: 101011

Signature:  
101011

(b)

```
y = 1;  
z = 1;  
print (x, y);  
print (x, z);  
x = 1;  
print (y, z);
```

Prints: 010111

Signature:  
110101

(c)

```
y = 1;  
x = 1;  
z = 1;  
print (x, z);  
print (y, z);  
print (x, y);
```

Prints: 111111

Signature:  
111111

(d)

But, for instance, **001001** is not allowed.

# Causal Consistency

---

- This model distinguishes between events that are “causally related” and those that are not.
- *If event  $B$  is caused or influenced by an earlier event  $A$ , then causal consistency requires that every other process see event  $A$ , then event  $B$ .*
- Operations that are not causally related are said to be *concurrent*.

# More on Causal Consistency

---

- A causally consistent data-store obeys this condition:
- *Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines (i.e., by different processes).*

P1:	W(x)a		W(x)c	
P2:		R(x)a	W(x)b	
P3:		R(x)a		R(x)c
P4:		R(x)a		R(x)b
			R(x)b	R(x)c

This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store. Note: it is assumed that  $W_2(x)b$  and  $W_1(x)c$  are concurrent.

# Another Causal Consistency Example

P1:	W(x)a		
P2:		R(x)a	W(x)b
P3:			R(x)b R(x)a
P4:			R(x)a R(x)b

(a) **incorrect**

P1:	W(x)a		
P2:			W(x)b
P3:			R(x)b R(x)a
P4:			R(x)a R(x)b

(b) **correct**

- a) **Violation of causal-consistency** – P2's write is related to P1's write due to the read on 'x' giving 'a' (all processes must see them in the same order).
- b) **A causally-consistent data-store**: the read has been removed, so the two writes are now *concurrent*. The reads by P3 and P4 are now OK.

# FIFO Consistency

---

- Defined as follows:

*Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.*

- This is also called “PRAM Consistency” – Pipelined RAM.
- The attractive characteristic of FIFO is that **it is easy to implement**. There are no guarantees about the order in which different processes see writes – except that two or more writes from a single process must be seen in order.



# FIFO Consistency Example

P1:  $\forall x (W(x) \rightarrow a)$

P2:            R(x)a        W(x)b        W(x)c

P3:  $R(x)b \quad R(x)a \quad R(x)c$

P4:  $R(x)a \quad R(x)b \quad R(x)c$

- A valid sequence of FIFO consistency events.
- Note that none of the consistency models studied so far would allow this sequence of events.

# Introducing Weak Consistency

---

- Not all applications need to see all writes, let alone seeing them in the same order.
- This leads to “Weak Consistency” (which is primarily designed to work with *distributed critical regions*).
- This model introduces the notion of a “synchronization variable”, which is used update all copies of the data-store.

# Weak Consistency Properties

---

- The three properties of Weak Consistency:
  1. Accesses to synchronization variables associated with a data-store are *sequentially consistent*.
  2. No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.
  3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

# Weak Consistency: What It Means

---

- So ...
- By doing a sync., a process can *force* the just written value out to all the other replicas.
- Also, by doing a sync., a process can be *sure* it's getting the most recently written value before it reads.
- In essence, the weak consistency models enforce consistency on a *group of operations*, as opposed to individual reads and writes (as is the case with strict, sequential, causal and FIFO consistency).

# Weak Consistency Examples

---

P1:	W(x)a	W(x)b	<span style="border: 1px solid red; padding: 0 2px;">S</span>
P2:		R(x)a	R(x)b <span style="border: 1px solid red; padding: 0 2px;">S</span>
P3:		R(x)b	R(x)a <span style="border: 1px solid red; padding: 0 2px;">S</span>

(a) before sync., any results are acceptable

P1:	W(x)a	W(x)b	S
P2:		S	R(x)a

Wrong!!

(b)

- a) A valid sequence of events for weak consistency. This is because P2 and P3 have yet to synchronize, so there's no guarantees about the value in 'x'.
- b) An invalid sequence for weak consistency. P2 has synchronized, so it cannot read 'a' from 'x' – it should be getting 'b'.

# Introducing Release Consistency

---

- Question: how does a weakly consistent data-store know that the sync is the result of a read or a write?
- Answer: It doesn't!
- It is possible to implement efficiencies if the data-store is able to determine whether the sync is a read or write.
- Two sync variables can be used, “acquire” and “release”, and their use leads to the “Release Consistency” model.

# Release Consistency

---

□ Defined as follows:

*When a process does an “acquire”, the data-store will ensure that all the local copies of the protected data are brought up to date to be consistent with the remote ones if needs be.*

*When a “release” is done, protected data that have been changed are propagated out to the local copies of the data-store.*

# Release Consistency Example

P1: Acq(L)    W(x)a    W(x)b    Rel(L)

P2:  $Acq(L) \quad R(x)b \quad Rel(L)$

P3:  $R(x)a$

- A valid event sequence for release consistency.
- Process P3 has not performed an *acquire*, so there are no guarantees that the read of 'x' is consistent. The data-store is simply not obligated to provide the correct answer.
- P2 does perform an *acquire*, so its read of 'x' is consistent.



# Release Consistency Rules

---

- A distributed data-store is “Release Consistent” if it obeys the following rules:
  1. Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
  2. Before a release is allowed to be performed, all previous reads and writes by the process must have completed.
  3. Accesses to synchronization variables are *FIFO consistent* (sequential consistency is not required).

# Introducing Entry Consistency

---

- A different twist on things is “Entry Consistency”. Acquire and release are still used, and the data-store meets the following conditions:
  1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
  2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
  3. After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

# Entry Consistency: What It Means

---

- So, at an *acquire*, all remote changes to guarded data must be brought up to date.
- Before a write to a data item, a process must ensure that no other process is trying to write *at the same time*.

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)
P2:			Acq(Lx)	R(x)a		R(y)NIL
P3:				Acq(Ly)		R(y)b

Locks associate with individual data items, as opposed to the entire data-store. Note: P2's read on 'y' returns NIL as no locks have been requested.

# Summary of Consistency Models

- a) Consistency models that do not use synchronization operations.
- b) Models that do use synchronization operations. (These require additional programming constructs, and allow programmers to treat the data-store *as if it is sequentially consistent*, when in fact it is not. They “should” also offer the best performance).

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done.
Release	Shared data are made consistent when a critical region is exited.
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

# Protocols – Dash Protocols, NUMA Multiprocessors, NUMA Algorithms

**Dr. Roshni Pradhan**  
**Assistant Professor**  
**School of Computer Engineering**  
**KIIT Deemed to be University**  
**Bhubaneswar**

# Directory Architecture for Shared memory (DASH)

- Distributed shared memory paradigm enables shared memory view of a loosely coupled distributed memory system. The DASH prototype developed at Stanford University is example of such systems.
- The DASH prototype belongs to the non uniform memory access (NUMA) class and makes use of the directory-based protocol for maintaining cache coherence.
- The DASH is a high-performance machine with single address space and coherent caches.
- The DASH architecture consists of a 2-D mesh network of clusters; each cluster consists of a set of processor elements (PEs) sharing a common communication channel. The cache coherence protocol is based on a four-level memory hierarchy protocol.

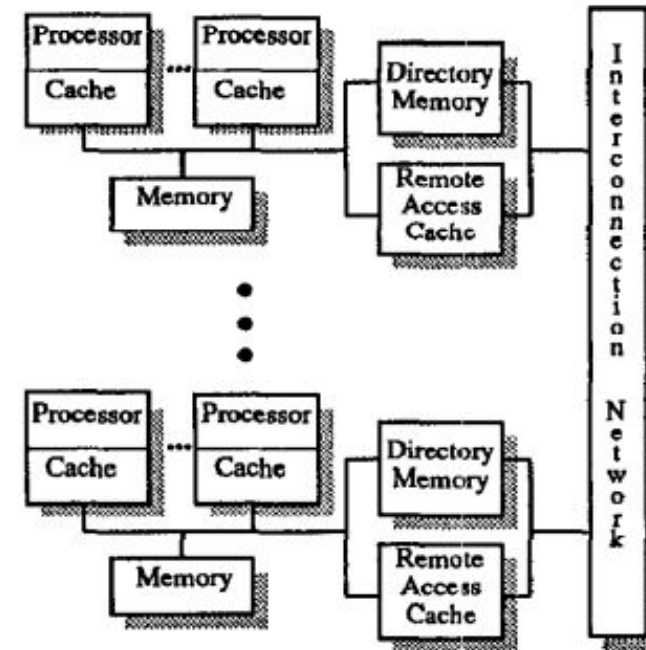


Figure 1: General architecture of DASH.

# DASH Cache Coherence Protocol

- The DASH coherence protocol is an invalidation-based ownership protocol.
- A memory block can be in one of three states as indicated by the associated directory entry:
  - (i) uncached-remote, that is not cached by any remote cluster
  - (ii) shared-remote, that is cached in an unmodified state by one or more remote clusters or
  - (iii) dirty-remote, that is cached in a modified state by a single remote cluster
- There are three primitive operations supported by the base DASH coherence protocol i.e read, readexclusive and write-back.

**Read Requests:** Memory read requests are initiated by processor load instructions. If the location is present in the processor's first-level cache, the cache simply supplies the data. If not present, then a cache fill operation must bring the required block into the firstlevel cache.

**Read-Exclusive Requests:** Write operations are initiated by processor store instructions. Data is written through the first-level cache and is buffered in a four word deep write-buffer. The second-level cache can retire the write if it has ownership of the line. Otherwise, a readexclusive request is issued to the bus to acquire sole ownership of the line and retrieve the other words in the cache block.

**Writeback Requests:** A dirty cache line that is replaced must be written back to memory. If the home of the memory block is the local cluster, then the data is simply written back to main memory. If the home cluster is remote, then a message is sent to the remote home which updates the main memory and marks the block uncached-remote.

# Non-uniform memory access (NUMA)

- Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing systems where memory access time depends on the memory location relative to the processor accessing it.
- In a NUMA architecture, multiple processors (or nodes) are connected to a shared memory pool.
- Each processor has its own local memory and can also access memory from other processors over a high-speed interconnect.
- However, accessing local memory is faster than accessing remote memory due to differences in latency.
- The primary motivation behind NUMA is to alleviate memory access bottlenecks in large-scale multiprocessing systems. As the number of processors increases in a system, the contention for accessing a shared memory pool also increases, leading to performance degradation.
- By partitioning memory into local and remote segments based on the proximity to processors, NUMA aims to reduce contention and improve overall system performance.
- NUMA architectures typically employ hardware or software mechanisms to manage memory access efficiently.



# Importance of NUMA

NUMA is essential to modern servers and mainframe environments for several reasons. This is because it paves the way for:

- Reduced memory access latency
- Improved performance
- Greater flexibility
- Enhanced parallelism
- Optimized cache utilization
- Adaptive resource management
- Fault tolerance
- High availability
- Compatibility with heterogeneous workloads
- Energy efficiency

# Applications of NUMA

NUMA architecture finds applications across various industries requiring scalable multiprocessing systems to handle complex tasks efficiently. Here are some use cases where NUMA-based solutions can be presented.

- **Enterprise computing:** In enterprise computing environments, NUMA architectures are widely used in servers and data centers to support mission-critical applications such as database management systems (DBMS), enterprise resource planning (ERP) software, and virtualization platforms.
- **High-performance computing (HPC):** In scientific research, engineering simulations, and computational modeling, NUMA models play a crucial role in high-performance computing (HPC) clusters and supercomputers.
- **Financial services:** NUMA frameworks are used in trading platforms, risk analysis systems, and algorithmic trading algorithms in the financial services industry.
- **Telecommunications:** NUMA systems are employed in network appliances, packet processing systems, and software-defined networking (SDN) controllers in telecommunications infrastructure. Telcos require fast and efficient data processing to handle network traffic routing, quality of service (QoS) management, and network function virtualization (NFV).
- **Healthcare and life sciences:** In healthcare and life sciences research, NUMA architectures support applications such as medical imaging analysis, genomic sequencing, and drug discovery.
- **Aerospace and defense:** NUMA is necessary for simulation and modeling systems for aircraft design, missile guidance systems, and radar signal processing. This makes it possible to simulate complex aerodynamic phenomena, analyze sensor data in real time, and optimize defense systems' performance.
- **Automotive:** NUMA architectures are utilized in vehicle design and testing, autonomous driving systems, and manufacturing process optimization. Automotive manufacturers rely on computational simulations to design and validate vehicle components, analyze crash scenarios, and optimize fuel efficiency.
- **Ecommerce and retail:** In the e-commerce and retail industry, NUMA mechanisms power recommendation engines, inventory management systems, and supply chain optimization algorithms.

# NUMA Algorithms

- **Linux NUMA Balancing:** In modern Linux systems, the kernel tracks memory access patterns and automatically moves memory pages closer to the processors accessing them, which is referred to as automatic NUMA balancing. It employs algorithms to determine whether migrating memory pages would improve performance and make memory allocation decisions accordingly.
- **Windows NUMA Scheduling and Allocation:** In Windows, NUMA-aware scheduling and memory management algorithms allow applications and the OS to optimize memory access patterns based on NUMA node locality, improving the performance of NUMA-enabled systems.
- **Java Virtual Machine (JVM) NUMA Awareness:** In memory-managed environments like the JVM, garbage collection algorithms can be NUMA-aware, ensuring that memory allocations and deallocations respect memory locality. This helps in reducing remote memory accesses and optimizing performance for memory-intensive applications.

# Question

1. “NUMA can be thought of as a microprocessor cluster in a box.”  
Justify this statement.
2. Compare NUMA, UMA and COMA shared memory concept.
3. What is cache coherence? What are most common mechanisms of ensuring coherency?

**THANK YOU**

# Introduction to Distributed Shared Memory (DSM)

Rakesh Kumar Rai  
Assistant Professor(1)  
School of Computer Engineering  
KIIT Deemed to be University,  
Bhubaneswar

# Introduction to DSM

- In distributed systems, Distributed Shared Memory (DSM) provides a way to make distributed memory across different computers look like a single shared memory.
- This allows processes running on different machines to share data as if they were working on the same physical memory, even though the data might be spread across multiple nodes.

# Shared-Variable DSM

- In shared-variable DSM, the system provides a shared memory abstraction where variables can be accessed directly by processes across different machines.
- These variables are stored in the memory of different machines, but from the perspective of the user or programmer, it looks like a single shared memory space.



# Advantages of Shared-Variable DSM

- Simplifies programming: Programmers can use familiar shared memory programming techniques without worrying about the underlying message-passing mechanisms.
- Improves scalability: Memory can be distributed across many machines, allowing larger systems with more memory capacity.

# Munin: Overview

- Munin is a shared-variable DSM system designed for multiprocessor systems.
- It allows processes on different processors to share memory, even when the memory is physically distributed across multiple nodes.

# Munin: Key Features

- Multiple consistency protocols: Munin uses different consistency protocols based on the type of data being accessed. For example, it can apply strict consistency for some variables and relaxed consistency for others to optimize performance.
- Lazy release consistency: Munin delays the propagation of updates until another process actually needs the data. This approach reduces unnecessary communication between nodes.
- Efficient communication: Munin reduces the amount of data sent between nodes by only sending updates when required.

# Munin: Use Case


- Munin is ideal for parallel computing environments where different parts of an application can be distributed across multiple processors.

# Midway: Overview

- Midway is another shared-variable DSM system, but it focuses on providing release consistency to applications, which ensures that updates to shared variables are propagated at specific synchronization points.

# Midway: Key Features

- Release consistency model: Midway uses the release consistency model, where updates to shared variables are delayed until certain synchronization operations (like locks or barriers) occur. This reduces unnecessary communication between nodes and improves performance.
- Weak consistency: By allowing more relaxed consistency, Midway can scale better in distributed systems with large numbers of processes.

- 
- Efficient synchronization: Midway provides efficient synchronization mechanisms, reducing the overhead of coordinating access to shared variables.

# Midway: Use Case

- Midway is suitable for distributed systems where communication costs are high, and it's beneficial to delay updates until absolutely necessary.





Thank You

# THREADS

By,

Dr Meghana G Raj, Associate Professor



# FAULT TOLERANCE IN DISTRIBUTED OPERATING SYSTEMS

- **Fault tolerance** means that a system can continue to function properly, even when some parts of the system fail. In a distributed system, where tasks are spread across multiple machines, this is critical because hardware or software failures can happen at any time. A fault-tolerant system has mechanisms to detect failures and recover from them without crashing the entire system.
- **Example:** Imagine an online shopping website that runs on multiple servers. If one server fails (due to a power outage, for instance), the system should automatically switch tasks to other working servers so that users can continue shopping without any disruptions. That's fault tolerance in action.



# SYSTEM FAILURE TYPES

- Failures in distributed systems are classified mainly into two types:
- **Fail-Silent Faults:** When a system experiences a **fail-silent fault**, it stops working silently (i.e., it stops processing tasks) but does not produce incorrect results or mislead other systems. It either functions correctly or stops functioning altogether.
- **Example:** Imagine a light bulb that either works perfectly or burns out completely. If it fails, it doesn't flicker or provide incorrect lighting; it just stops working. Similarly, in a distributed system, a server with a fail-silent fault either works properly or just stops without creating any confusion.
- **Byzantine Faults:** A **Byzantine fault** is more complicated because the system or component doesn't just fail silently. Instead, it behaves unpredictably. It could give wrong answers, provide partial data, or cause confusion for the rest of the system. Byzantine faults are difficult to detect because the faulty system might still appear to be running normally to others.
- **Example:** Imagine a taxi dispatch system where one taxi shows it's going to Location A, but actually drives to Location B. The other taxis and dispatchers have no idea what the problem is and are confused by the misleading information. This unpredictability makes Byzantine faults tricky to handle.



# SYNCHRONOUS VS. ASYNCHRONOUS SYSTEMS

- **Synchronous Systems**: In a synchronous system, there is a known and fixed time limit for communication and task execution. All parts of the system follow the same clock, so there's predictability in when messages are delivered and when processes are completed.
- Example: Think of a school where every class starts and ends at a specific time, and all teachers and students follow the same schedule. Everyone knows exactly when things will happen.
- **Asynchronous Systems**: In an asynchronous system, there are no fixed time limits. Processes and messages can take variable amounts of time to execute or deliver, and there's no guarantee of when tasks will be completed.
- Example: Imagine a postal service where packages arrive whenever they are delivered, without any fixed timeline. Sometimes they arrive in one day, sometimes in one week, and there's no strict time for when things should happen.



## ■ Key Differences:

1. **Timing guarantees:** Synchronous systems have strict time limits, while asynchronous systems don't.
2. **Complexity:** Synchronous systems are easier to design due to predictability, while asynchronous systems are harder due to uncertainty.
3. **Fault tolerance:** Asynchronous systems are more fault-tolerant because they expect delays, while synchronous systems might assume something is wrong if a task takes too long.
4. **Communication:** In synchronous systems, communication is reliable and timely, whereas in asynchronous systems, there can be unpredictable delays in communication.
5. **Examples:** Synchronous systems include real-time systems like flight control; asynchronous systems include most internet-based systems where delays are common.



## ■ **Redundancy**

- **Redundancy** involves having multiple copies of the same data or processes in case of failure. By duplicating critical parts of a system, it ensures that even if one part fails, the other can take over.
- **Degree of replication:** This refers to how many backups or replicas are made. For instance, in a cloud storage system, data might be replicated across three different servers so that even if one server fails, the data is still accessible from the other two.
- **Performance in the absence of faults:** When there are no failures, performance is generally optimal because all systems work together smoothly.
- **Performance when faults occur:** When a failure happens, redundant systems help maintain performance, though there might be a small delay as the system switches to backup components.
- **Example:** In an aircraft, there are usually three altimeters (instruments that measure altitude). If one fails, the other two provide the correct altitude, ensuring safety through redundancy.



- **Fault Tolerance Using Active Replication**
  - In **active replication**, multiple copies of the same process run simultaneously on different machines. If one machine fails, the others are already running the same process, so the system keeps working without interruption.
  - **Example:** Think of an emergency service call center where every call is routed to several operators at once. If one operator's phone breaks down, other operators can answer the call. This ensures that the service is always available.
- **Triple Modular Redundancy (TMR)**
  - **TMR** is a fault-tolerance mechanism where three identical systems process the same tasks, and a majority vote is taken. If one system fails, the other two can outvote the faulty one and provide the correct result.
  - **Example:** In critical systems like spacecraft control, TMR might be used to ensure safety. Three identical sensors measure the same data, and if one gives a wrong value, the system uses the value reported by the other two.
- **State Machine Approach and Active Replication**
  - The **state machine approach** assumes that every process in a system behaves like a finite state machine, meaning it follows a set of rules to move between states. Each replica of a process starts from the same state and processes the same sequence of events.
  - **Example:** Think of a vending machine where each step (insert money, select item, deliver item) follows a strict sequence. If you have several vending machines (replicas) following the same steps, even if one breaks down, the others will deliver the same outcome.





## ■ **Primary-Backup Protocol**

- In the **primary-backup protocol**, one machine (the primary) does all the work, and another machine (the backup) stays idle but keeps an updated copy of the data. If the primary fails, the backup takes over.
- **Example:** Imagine a bank's ATM system where one server handles all the transactions. Another server mirrors the data from the first one but does not process anything. If the first server crashes, the second one immediately takes over, continuing to process transactions without disruption.

## ■ **Agreement in Fault-Tolerant Systems**

- In fault-tolerant systems, processes must agree on a course of action, even if some of them are faulty. Achieving agreement becomes challenging in the presence of faults, especially Byzantine faults, where faulty processes may behave maliciously or unpredictably.

## ■ **Two Army Problem**

- The **Two Army Problem** is a theoretical communication issue where two groups (armies) need to coordinate an attack but can't fully trust the reliability of their messages. One army sends a message to the other confirming the time of attack, but since messages may get lost or delayed, neither army can be sure that the message was received, creating uncertainty.
- **Example:** If two taxi drivers need to meet at a location but can only communicate through a faulty GPS app, they can never be sure the other driver received their confirmation, leading to a lack of coordination.



- **Byzantine Generals Problem**
- The **Byzantine Generals Problem** is a famous problem in distributed systems where multiple generals (nodes) need to agree on a plan of action (e.g., attack or retreat). However, some generals may be traitors who send false or conflicting information to disrupt coordination. The challenge is for the loyal generals to reach a consensus despite the presence of faulty or malicious generals.
- **Example:** Imagine a group of ride-sharing drivers coordinating their routes. Some drivers (malicious or faulty) may give wrong directions, confusing others. The goal is for honest drivers to agree on the correct route, despite the misleading information from the faulty ones.
- The solution to this problem requires a way to detect and ignore the faulty generals' information to ensure that the majority of honest participants agree on the correct plan.
- These concepts are essential for understanding how distributed systems handle failures and ensure reliability

