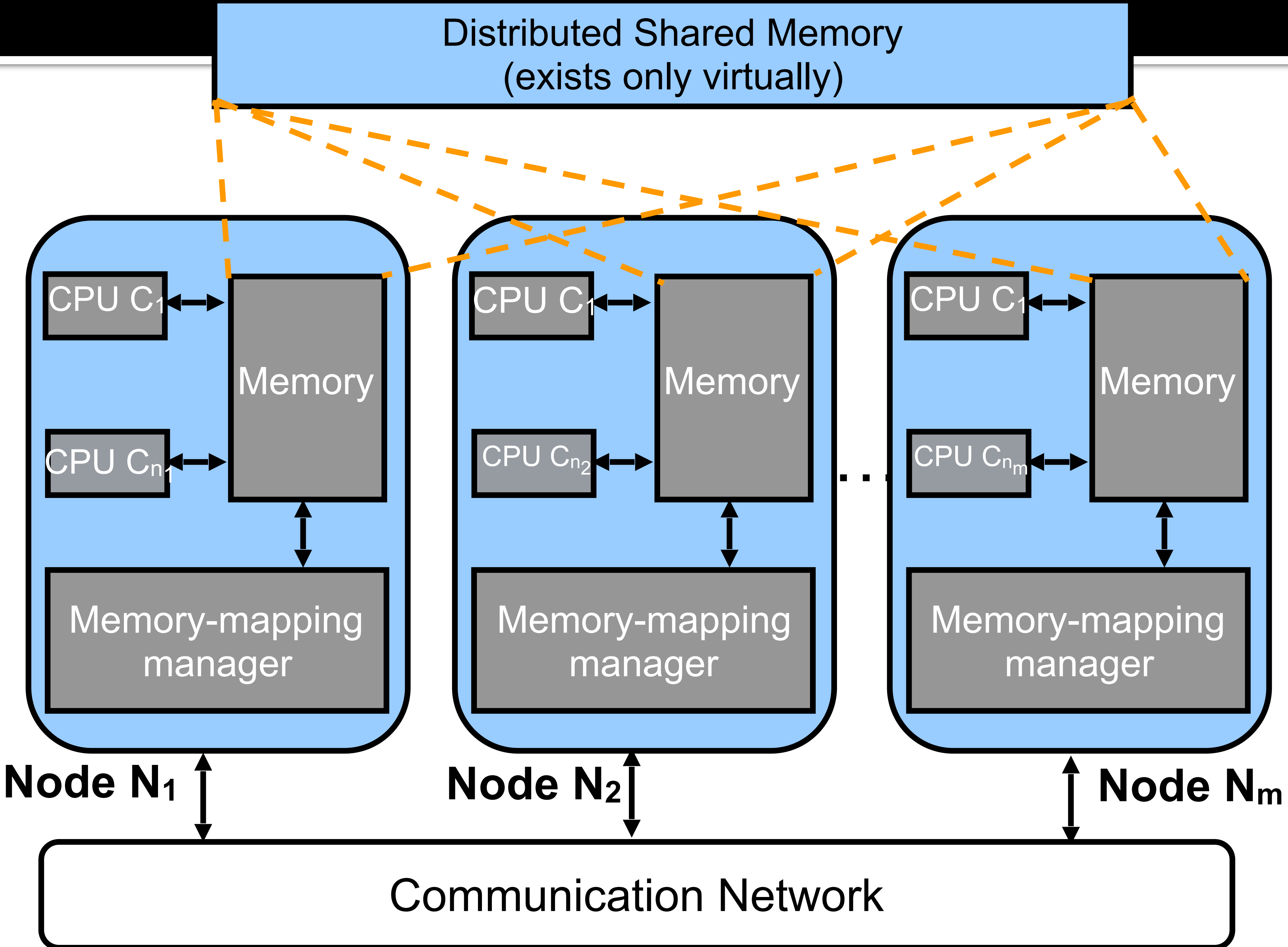
Advanced Operating Systems: Distributed Shared Memory

Motivation

- RPC allows us to pass messages to the processes in the distributed systems.
- RMI allows us to call procedures in the distributed systems.
- We used to have shared memory in uni-processor systems to share data between process.
- It is popular to use shared-memory in tightly-coupled multi-processor systems.
- How about loosely coupled distributed systems?

Distributed Shared Memory (DSM)



Advantages of DSM

- Simpler Abstraction
  - Programming distributed memory machines
  - Message passing models is tedious and error prone.
  - Under RPC and message passing, it is difficult to pass context-related data or complex data structures.
- Better Portability of Distributed Application Programs
  - Consistent access protocol makes it easier to transit from sequential applications to distributed applications.
  - Migrating shared-memory multiprocessor applications to distributed systems with distributed shared memory is seamless.
- Better Performance of Some Applications
  - Locality of Data
  - On-demand data movement
  - Larger memory space
- Flexible Communication Environment
- Ease of Process Migration

Design and Implementation Issues of DSM

- Granularity: block vs. page
- Structure of shared-memory space
- Memory coherence and access synchronization (consistence)
- Data location and access
- Replacement strategy
- Thrashing
- Heterogeneity

Coherence vs. Consistency

- Coherence concerns only ***one*** memory location
- Consistency concerns for ***all*** locations
- A memory system is coherence if
  - it can serialize all operations to that location
    - operations performed by any core appear in program order.
  - it reads return values written by last store to that location.
- A memory system is consistent if
  - if follows the rules of its memory model
    - operations on memory location appears in some defined order.

Coherence vs. Consistency

- Name a few coherence protocol:
  - **Snooping**: snooping is a process where the individual caches monitor address lines for accesses to memory locations that they have cached. When a write operation is observed, the cache controller invalidates its own copy of the snooped memory location.
  - **Snarfing**: a cache controller watches both address and data in an attempt to update its own copy of a memory location when a second master modifies a location in main memory. When a write operation is observed to a location that a cache has a copy of, the cache controller updates its own copy of the snarfed memory location with the new data.
- Name a few consistency protocol:
  - **Strict consistency**: if a process reads any memory location, the value returned by the read operation is the value written by the most recent write operation to that location.
  - **Sequential consistency**
  - **Processor consistency**

Coherent but not consistent

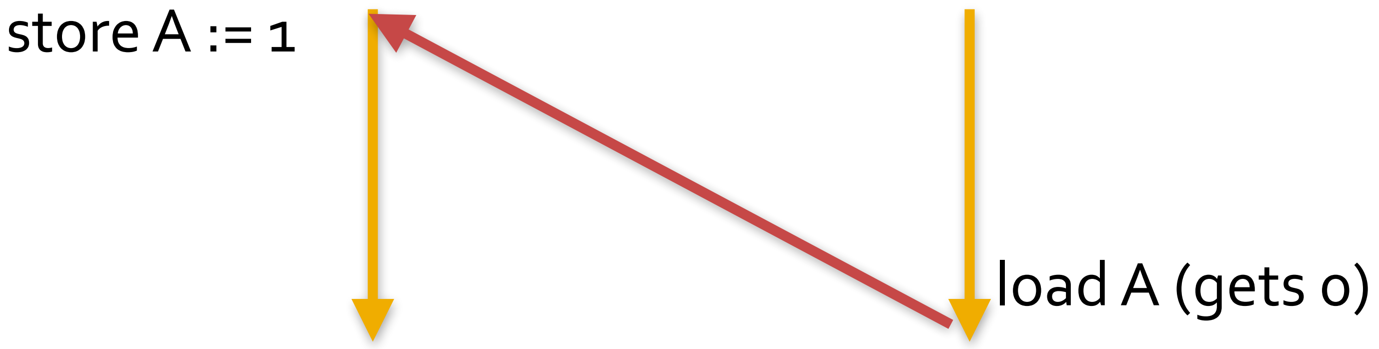
- Can you find a memory trace which is coherent but not consistent?

```
initially A=B=0
process 1          process 2
store A := 1        load B (gets 1)
store B := 1        load A (gets 0)
```

Coherent but not consistent

- Can you find a memory trace which is coherent but not consistent?

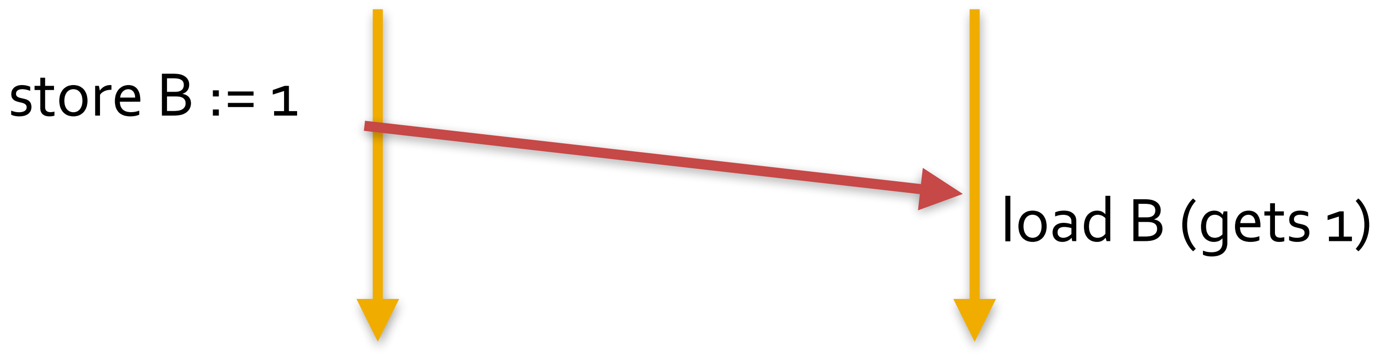
```
initially A=B=0
process 1            process 2
                      load A (gets 0)
store A := 1
```



Coherent but not consistent

- Can you find a memory trace which is coherent but not consistent?

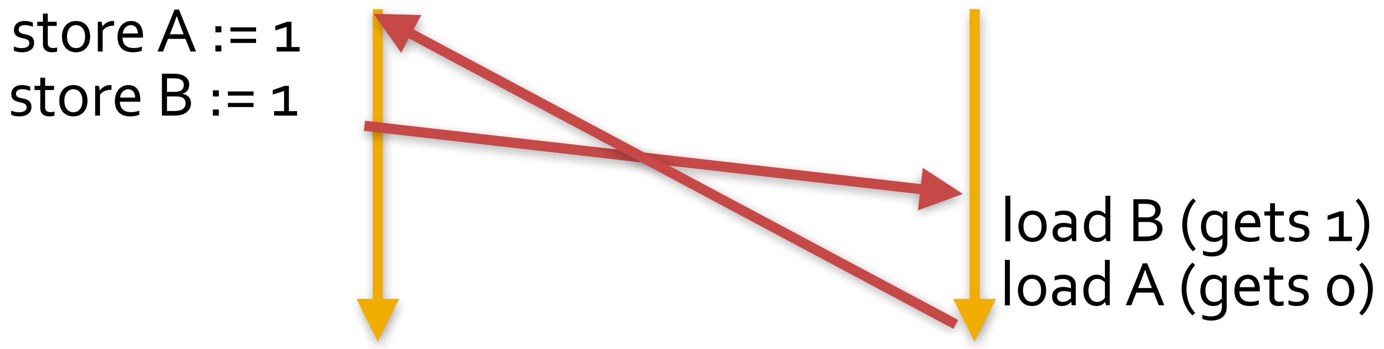
initially A=B=0	
process 1	process 2
store B := 1	
	load B (gets 1)



Coherent but not consistent

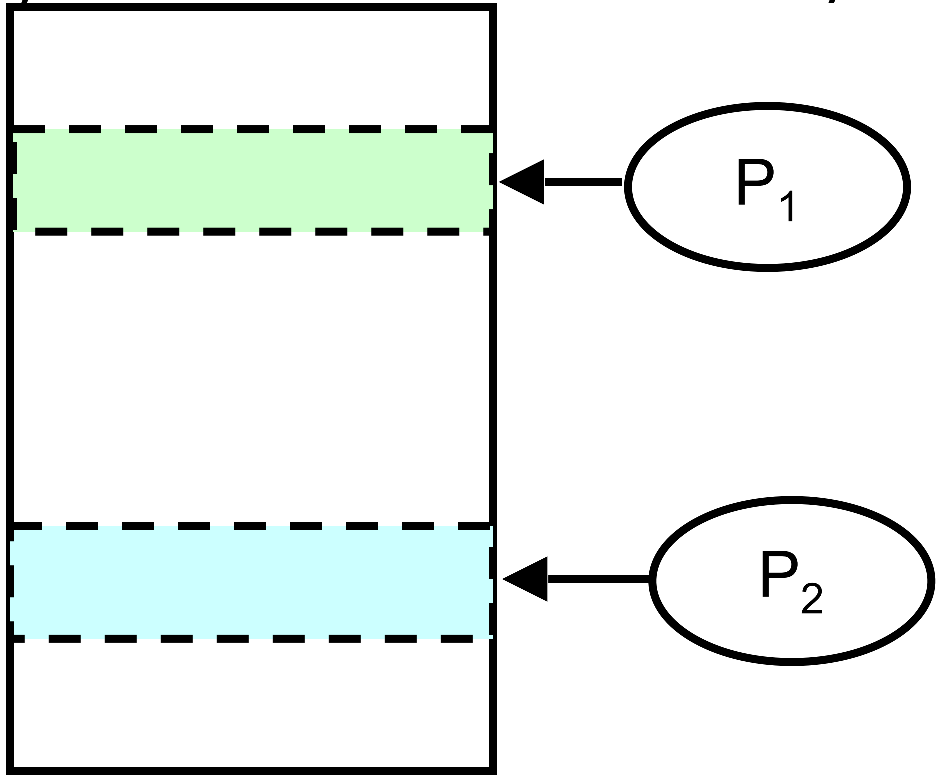
- Can you find a memory trace which is coherent but not consistent?

```
initially A=B=0
process 1          process 2
store A := 1        load B (gets 1)
store B := 1        load A (gets 0)
```



Granularity – How to select block size

- Block: the unit for transmitting data.
- Trade-off: network traffic vs. parallelism
- What's the difference between multi-processor system and distributed systems in terms of memory access?
- Factors to consider:
  - Paging overhead
  - Directory size
  - Thrashing
  - False sharing



One data block

Using page size as block size

- The system can use existing page fault schemes.
- The system can use existing access right control.
- If a page can be fitted into a packet, page sizes do not impose undue communication overhead.
- A page size is a suitable data entity with respect to memory contention.

Structure of Shared-Memory Space

- Structure: the abstract view of the shared-memory space
  - One may see the DSM as a storage of words and
  - The other may see the DSM as a storage of data objects.
- It is related to the choice of block size.
- Three common structures:
  - No structuring
    - Fixed grain size for all applications
    - Easier to choose any suitable page size as the unit of sharing
  - Structuring by data type
    - Variable grain size
    - Complicated design and implementation
  - Structuring as a database
    - Tuple space: memory ordered by their content.
    - Accessed by specifying the number of their fields and their values via special access functions
- How does the type of structure affect the implementation of your systems?

Consistency Models

- Consistency models: the degree of consistency that has to be maintained
- Ongoing researches: relax the requirements to a greater degree.
- Example of different consistency models:

看股票　　　看開票

  - Which one aims on ordering?
  - Which one aims on results?

Consistency Models

- Stronger consistency model vs. weaker consistency model
- Available models:
  - Strict consistency model
  - Sequential consistency model
  - Causal consistency model
  - Pipelined random-access memory consistency model
  - Processor consistency model
  - Weak consistency model
  - Release consistency model

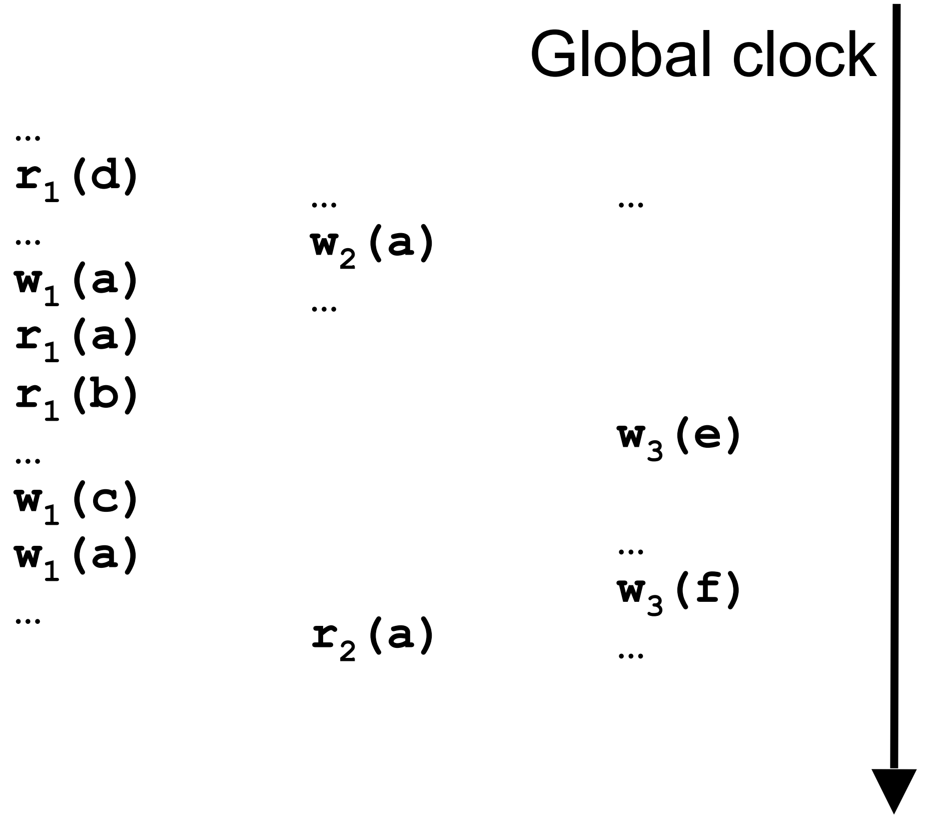
Strict Consistency Model

- The value returned by a read operation on a memory address is always the same as the value written by the <u>most recent</u> write operation to that address.
- All writes instantaneously become visible to all processes.
- What you need:
  - read/write operations must be correctly ordered
  - an absolute global clock

Consistency Models – Strict Consistency

Node N_1	Node N_2	Node N_3

```
{                    {                    {
  …                    …                    …
  a=d                  a=10                 e=foo()
  c=a+b                …                    f=bar()
  a=4                  print(a)             …
  …                  }                    }
}
```

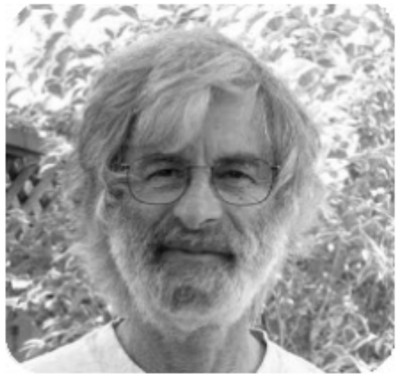


Strict Consistency Model

Node N_1	Node N_2	Node N_3
…	…	…
$r_1(d)$	$r_1(d)$	$r_1(d)$
$w_2(a)$	$w_2(a)$	$w_2(a)$
$w_1(a)$	$w_1(a)$	$w_1(a)$
$r_1(a)$	$r_1(a)$	$r_1(a)$
$r_1(b)$	$r_1(b)$	$r_1(b)$
$w_3(e)$	$w_3(e)$	$w_3(e)$
$w_1(c)$	$w_1(c)$	$w_1(c)$
$w_1(a)$	$w_1(a)$	$w_1(a)$
$w_3(f)$	$w_3(f)$	$w_3(f)$
$r_2(a)$	$r_2(a)$	$r_2(a)$
…	…	…

Sequential Consistency Model

- It was proposed by Lamport in '79.
- All processes see the same order of all memory access operations on the shared memory.
  - The orders seen by processes must be the same but
  - They are not necessary to be equal to the **EXACT** orders.
- The sequential consistency model does not guarantee that a read operation on a particular memory address always returns the same value as written by the most recent write operation to that address.
- Running a program twice may not give the same result in the absence of explicit synchronization operations.
- A sequential consistent memory provides *one-copy/single-copy* semantics because all the processes sharing a memory location always see the same contents.



LESLIE LAMPORT

United States – 2013

CITATION

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

SHORT ANNOTATED BIBLIOGRAPHY | ACM DL AUTHOR PROFILE | ACM TURING AWARD LECTURE VIDEO | RESEARCH SUBJECTS

PHOTOGRAPHS

BIRTH:

7 February 1941 in New York, New York

EDUCATION:

Bronx High School of Science (1957); B.S. (Massachusetts Institute of Technology, Mathematics, 1960); M.S (Brandeis University, Mathematics, 1963); PhD (Brandeis University, Mathematics, 1972).

EXPERIENCE:

Massachusetts Computer Associates, 1970-1977; SRI International, 1977-1985; Digital Equipment Corporation and Compaq, 1985-2001; Microsoft Research, from 2001.

HONORS AND AWARDS:

If we could travel back in time to 1974, perhaps we would have found Leslie Lamport at his busy local neighborhood bakery, grappling with the following issue. The bakery had several cashiers, but if more than one person approached a single cashier at the same time, that cashier would try to talk to all of them at once and become confused. Lamport realized that there needed to be some way to guarantee that people approached cashiers one at a time. This problem reminded Lamport of an issue which has been posed in an earlier article by computer scientist Edsger Dijkstra on another mundane issue: how to share dinner utensils around a dining table. One of the coordination challenges was to guarantee that each utensil was used by at most one diner at a time, which came to be generalized as the ***mutual exclusion*** *problem, exactly the challenge Lamport faced at the bakery.*

One morning in 1974, an idea came to Lamport on how the bakery customers could solve mutual exclusion among themselves, without relying on the bakery for help. It worked roughly like this: people choose numbers when they enter the bakery, and then get served at the cashier according to their number ordering. To choose a number, a customer asks for the number of everyone around her and chooses a number higher than all the others.

This simple idea became an elegant algorithm for solving the mutual exclusion problem without requiring any lower-level indivisible operations. It also was a rich source of future ideas, since many issues had to

Consistency Models – Sequential Consistency

Node N_1	Node N_2	Node N_3

`{`
`  …`
`  a=d`
`  c=a+b`
`  a=4`
`  …`
`}`
	`{`
	`  …`
	`  a=10`
	`  …`
	`  print(a)`
	`}`
		`{`
		`  …`
		`  e=foo()`
		`  f=bar()`
		`  …`
		`}`

…		
$r_1(d)$	…	…
…	$w_2(a)$	
$w_1(a)$	…	
$r_1(a)$		
$r_1(b)$		
		$w_3(e)$
…		
$w_1(c)$		
$w_1(a)$		…
		$w_3(f)$
…	$r_2(a)$	
		…

Sequential Consistency Model

Global clock

Node N_1	Node N_2	Node N_3
…	…	…
$r_1(d)$	$r_1(d)$	$r_1(d)$
$w_1(a)$	$w_1(a)$	$w_1(a)$
$w_2(a)$	$w_2(a)$	$w_2(a)$
$r_1(a)$	$r_1(a)$	$r_1(a)$
$r_1(b)$	$r_1(b)$	$r_1(b)$
$w_3(e)$	$w_3(e)$	$w_3(e)$
$w_1(c)$	$w_1(c)$	$w_1(c)$
$w_1(a)$	$w_1(a)$	$w_1(a)$
$w_3(f)$	$w_3(f)$	$w_3(f)$
$r_2(a)$	$r_2(a)$	$r_2(a)$
…	…	…

What're the difficulties of implementing consistency model?

- Each node/process needs to know which instructions are issued other nodes/processes.
  - Communications or synchronizations are required among the nodes/processes.
  - Communications/synchronizations will slow down or block the progress.
- Consequently, the performance of the systems become poor.
  - When the number of nodes/processes increase, the penalty increases (exponentially).

Further relaxing the model to avoid communication overhead

- The outcome of a sequence of memory operations depend on
  - execution order and
  - what else?

```
a=0;b=0;
a=1;
b=a+2;
print("a: %d, b:%d\n", a, b);
```

```
a=0;b=0;
b=a+2;
a=1;
print("a: %d, b:%d\n", a, b);
```

Causality 因果關係

Further relax the model

- The outcome of a sequence of memory operations are related to
  - execution order and
  - what else?

```
a=0;b=0;
a=1;
b=2;
print("a: %d, b:%d\n", a, b);
```

```
a=0;b=0;
b=2;
a=1;
print("a: %d, b:%d\n", a, b);
```

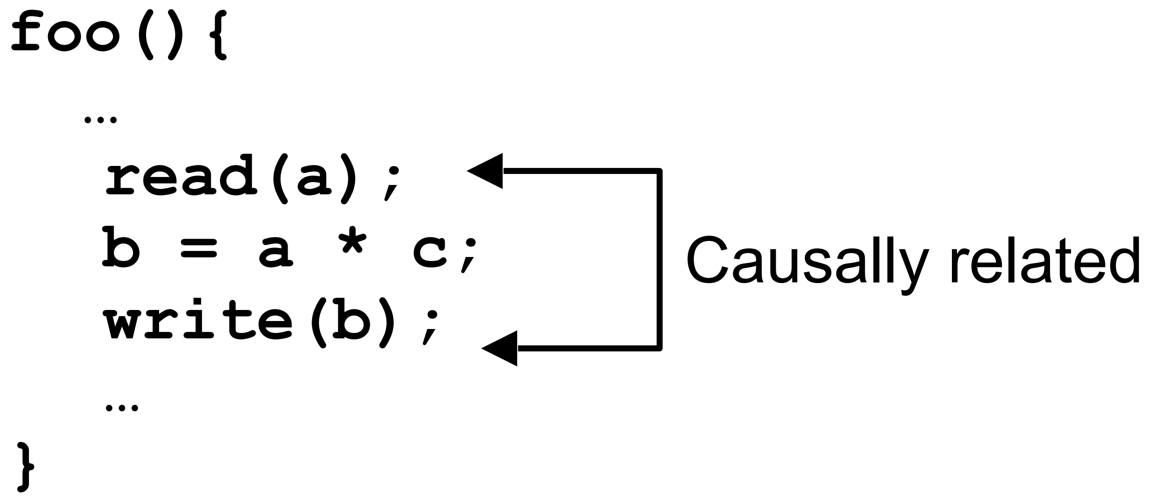
When there is no causality, the execution order has no effects.

Causally Related

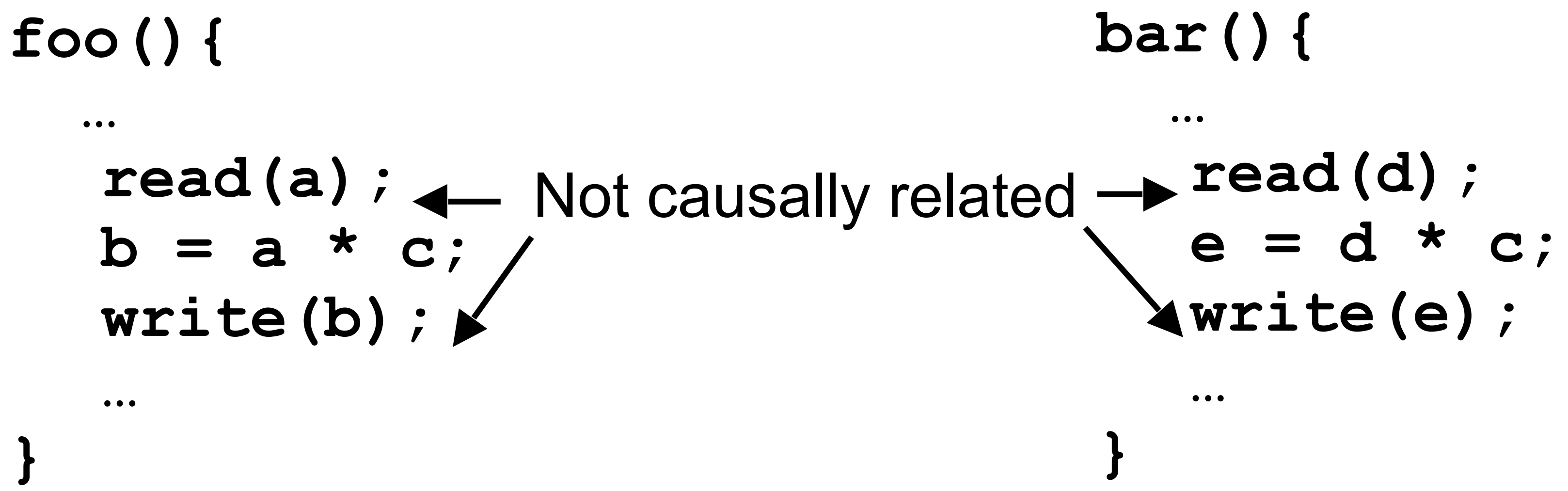
因果關係

與前面的模型主要差在個體間序列可以不同

A memory reference operation (read/write) is said to be potentially causally related to another memory reference operation if the second one might have been influenced in any way by the first one.



```
foo(){
  …
   read(a);
   b = a * c;
   write(b);
   …
}
```



```
foo(){
  …
   read(a);
   b = a * c;
   write(b);
   …
}
```

```
bar(){
  …
   read(d);
   e = d * c;
   write(e);
   …
}
```

Causal Consistency Models

- It is proposed by Hutto and Ahamad in '90.
- In the Causal consistency model,
  - all processes see only those memory reference operations in the same order that are potentially causally related,
  - memory reference operations that are not causally related may be seen by different processes in different orders.
- A shared memory system is said to support the causal consistency model if all write operations that are potentially causally related are seen by all processes in the same (correct) order.
  - Suppose W2 is causally related to W1, i.e., W2 depends on the results of W1.
  - Only (W1, W2) is correct. (W2, W1) is not.

Consistency Models – Causal Consistency

Node N_1	Node N_2	Node N_3
`{ … a=d c=a+b a=4 … }`	`{ … a=10 … print(a) }`	`{ … e=foo() f=bar() … }`

```
Node N1          Node N2          Node N3
{                {                {
  …                …                …
  a=d              a=10             e=foo()
  c=a+b            …                f=bar()
  a=4              print(a)         …
  …              }                }
}
```

Node N_1	Node N_2	Node N_3
…		
$r_1(d)$	…	…
…	$w_2(a)$	
$w_1(a)$	…	
$r_1(a)$		
$r_1(b)$		
		$w_3(e)$
…		
$w_1(c)$		…
$w_1(a)$		$w_3(f)$
…	$r_2(a)$	…

Causal Consistency Model

Global clock

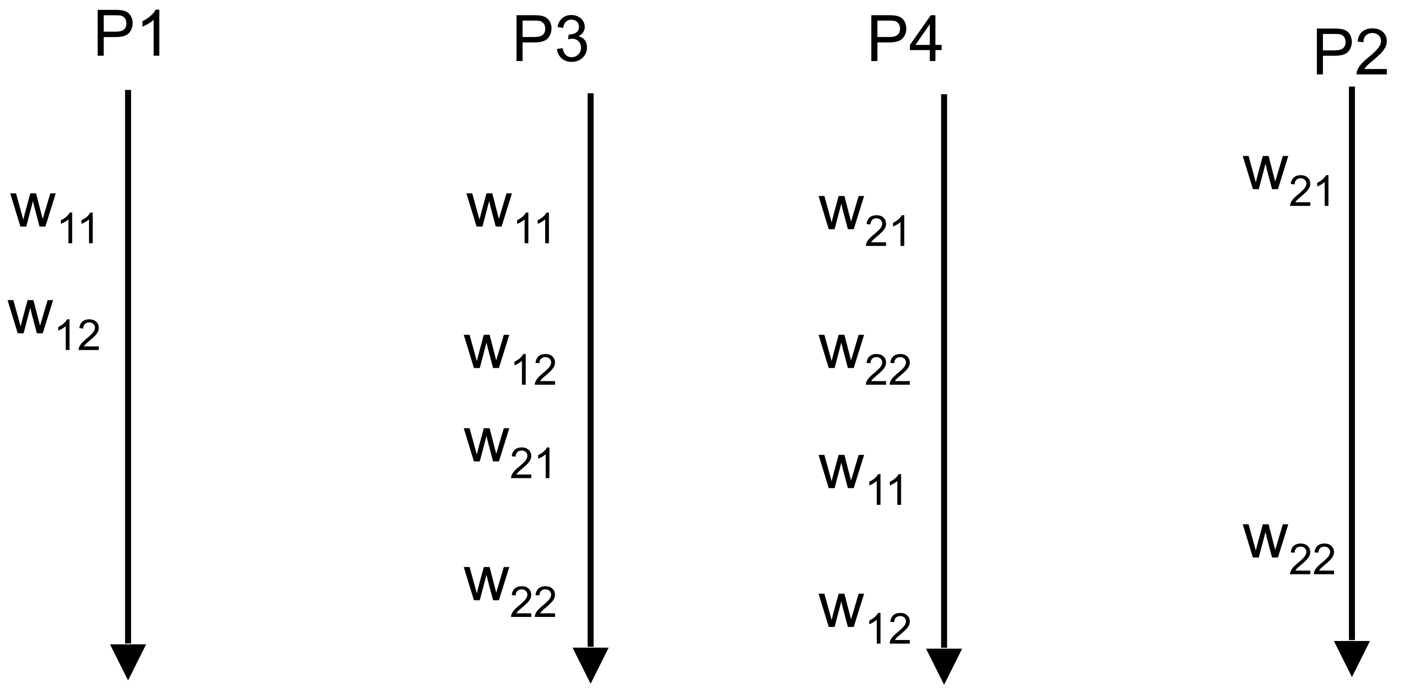
錯誤範例

Node N_1	Node N_2	Node N_3
…	…	…
$r_1(d)$	$r_1(d)$	$r_1(d)$
$w_3(e)$	$w_2(a)$	$w_1(a)$
$w_2(a)$	$w_3(e)$	$w_2(a)$
$w_1(a)$	$w_1(a)$	$r_1(a)$
$r_1(a)$	$r_1(a)$	$r_1(b)$
$r_1(b)$	$r_1(b)$	$w_3(e)$
$w_1(c)$	$w_1(c)$	$w_3(f)$
$w_3(f)$	$w_1(a)$	$w_1(c)$
$w_1(a)$	$w_3(f)$	$w_1(a)$
$r_2(a)$	$r_2(a)$	$r_2(a)$
…	…	…

Pipelined Random-Access Consistency Model

- It is proposed by Lipton and Sandberg in ’88.
- PRAM Consistency Model:
  - All write operations performed by a single process are seen by all other processes in the order in which they were performed as if all the write operations performed by a single process in a pipeline.
  - Write operations performed by different processes may be seen by different processes in different orders.

適用沒有因果關係
可以不用 lock 解決 false sharing
false sharing (page lock for processes accessing a same page)



- PRAM Consistency Model is simple and easy to implement.

Consistency Models - PRAM

n1, n2'a are different

Node N_1	Node N_2	Node N_3
{ … a=d c=a+b a=4 … }	{ … a=10 … print(a) }	{ … e=foo() f=bar() … }

Global clock

Node N_1	Node N_2	Node N_3
…		
r_1(d)		
…	…	…
	w_2(a)	
…	…	
w_1(a)		
r_1(a)		
r_1(b)		
		w_3(e)
…		
w_1(c)		
w_1(a)		…
		w_3(f)
…	r_2(a)	
		…

同一行的動作相對位置要相同，主要判斷 write

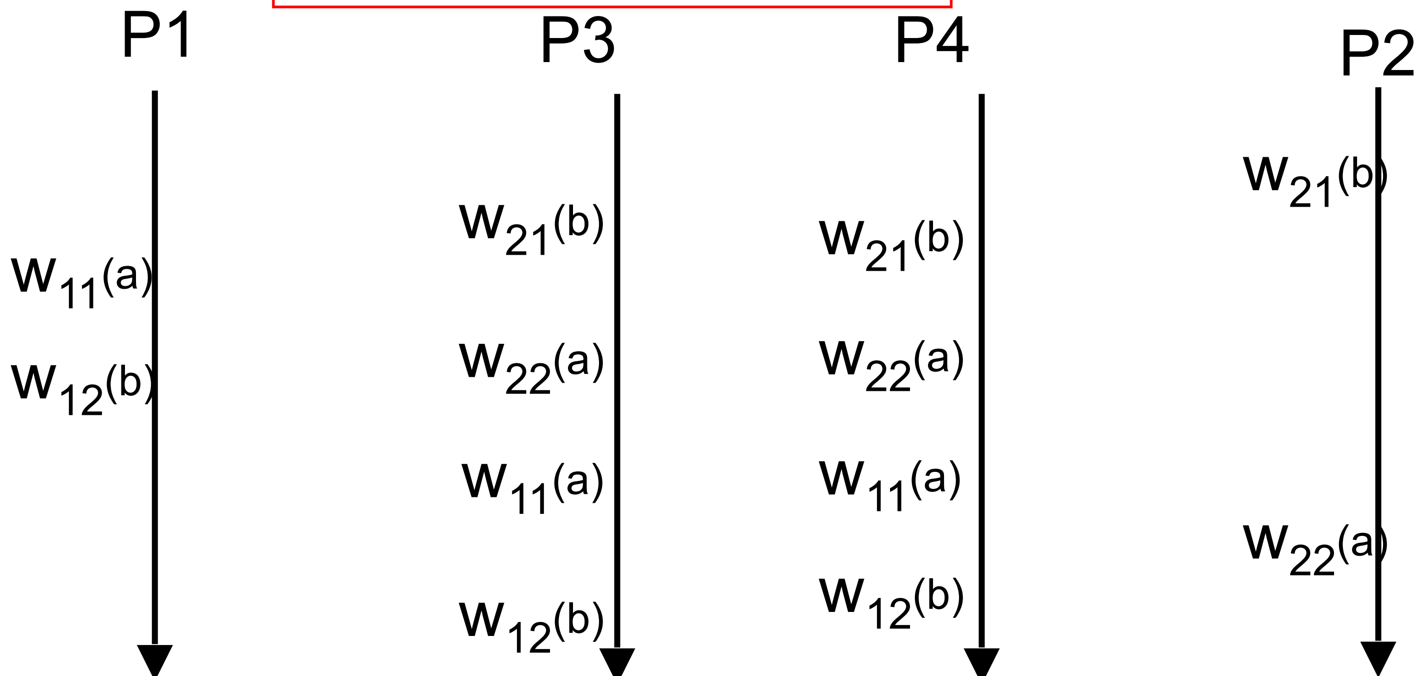
Pipelined Random-Access Memory Consistency Model

Node N_1	Node N_2	Node N_3
…	…	…
r_1(d)	r_1(d)	r_1(d)
w_3(e)	w_2(a)	w_1(a)
w_3(f)	w_1(a)	r_1(a)
w_2(a)	r_1(a)	r_1(b)
w_1(a)	r_1(b)	w_3(e)
r_1(a)	w_1(c)	w_3(f)
r_1(b)	w_1(a)	w_2(a)
w_1(c)	w_3(e)	w_1(c)
w_1(a)	w_3(f)	w_1(a)
r_2(a)	r_2(a)	r_2(a)
…	…	…

Consistency Models – Processor Consistency Model

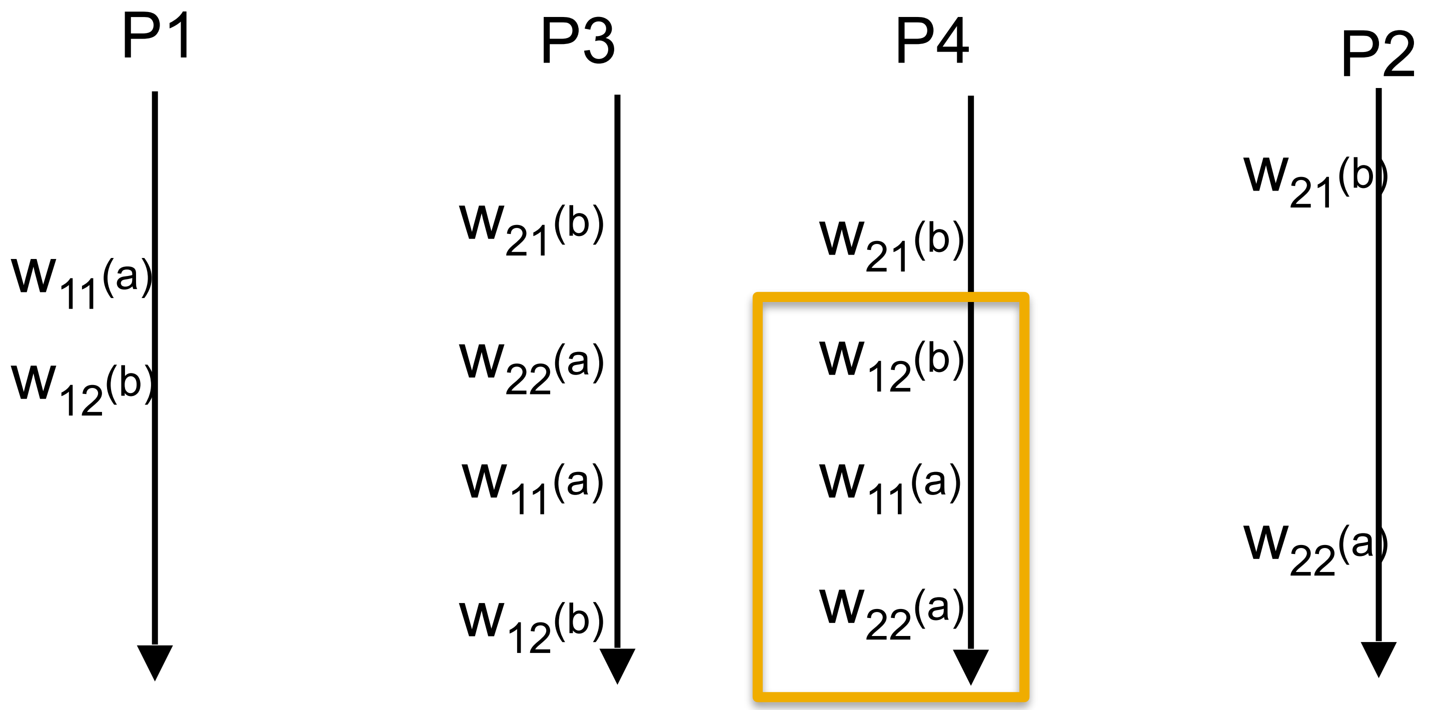
- Proposed by Goodman in ’89.
- Adding coherent and adheres to the PRAM consistency model.
- Memory coherent:
  - for any memory location all processes agree on the same order of all WRITE operations to that location.
  - The WRITE operations on different memory location can be in different orders.

W21, W22次序顛倒也沒關係



Consistency Models – Processor Consistency Model

- How about this sequence?



Consistency Models – Processor Consistency Model

Node N_1	Node N_2	Node N_3

```
{
  …
  a=d
  c=a+b
  a=4
  …
}
```
	```
	{
	  …
	  a=10
	  …
	  print(a)
	}
	```
		```
		{
		  …
		  e=foo()
		  f=bar()
		  …
		}
		```

Global clock

Node N_1	Node N_2	Node N_3
…		
$r_1(d)$	…	…
…	$w_2(a)$	
$w_1(a)$	…	
$r_1(a)$		
$r_1(b)$		
		$w_3(e)$
…		
$w_1(c)$		…
$w_1(a)$		$w_3(f)$
…	$r_2(a)$	…

Node N_1　Node N_2　Node N_3

Processor Consistency Model

Memory coherences: any memory location all processes agree on the same order of all write operations to that location.

只需判斷 a　　判斷w為主

Node N_1	Node N_2	Node N_3
…	…	…
$r_1(d)$	$r_1(d)$	$r_1(d)$
$w_3(e)$	$w_2(a)$	$w_2(a)$
$w_3(f)$	$w_1(a)$	$w_1(a)$
$w_2(a)$	$r_1(a)$	$r_1(a)$
$w_1(a)$	$r_1(b)$	$r_1(b)$
$r_1(a)$	$w_1(c)$	$w_1(c)$
$r_1(b)$	$w_1(a)$	$w_1(a)$
$w_1(c)$	$w_3(e)$	$r_2(a)$
$w_1(a)$	$w_3(f)$	$w_3(e)$
$r_2(a)$	$r_2(a)$	$w_3(f)$
…	…	…

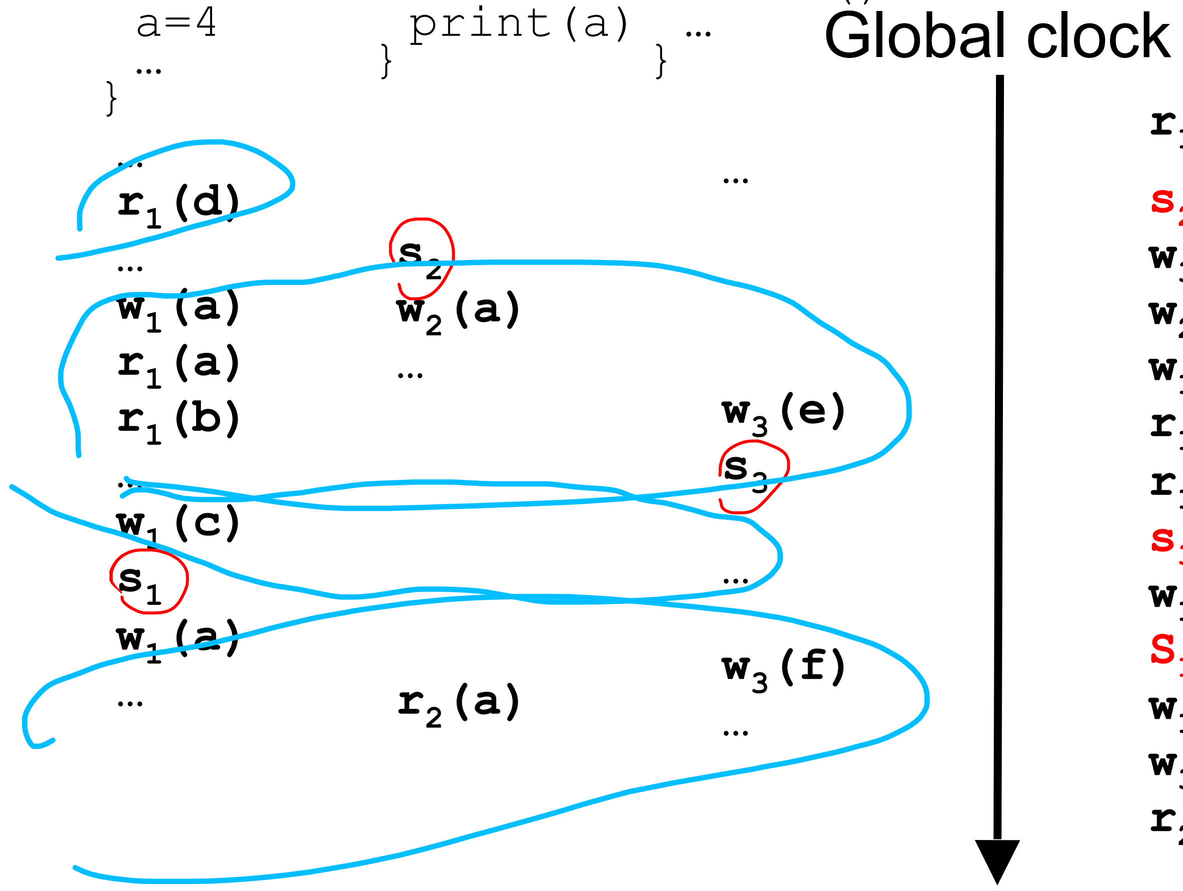
Consistency Models – Weak Consistency Model

- Observations by Dubois et al. [1988]:
  - Not necessary to show the change done by every write operation.
  - Isolated access to shared variables are rare.
- Better performance can be achieved if consistency is enforced on a group of memory reference operations rather than on individual memory reference operations. **programmer defined**
- A synchronization variable is used to propagate all writes to other machines, and to perform local updates with regard to changes to global data that occurred elsewhere in the distributed system.
- The properties of weak consistency: **可以針對locality，累積一定量的write再update**
  - Accesses to synchronization variables are sequentially consistent.
  - No access to a synchronization variable is allowed to be performed until all previous writes have been completed everywhere. -> To propagate the write before end.
  - No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed. -> To accept all the updates before start.

ex. join() in multiprocessing

Consistency Models – Weak Consistency Model

```
Node N1        Node N2        Node N3
{              {              {
  …              …              …
  a=d            s              e=foo()
  c=a+b          a=10           s
  s              …              f=bar()
  a=4            print(a)       …
  …            }              }
}
```



Weak Consistency Model

Node N_1	Node N_2	Node N_3
$r_1(d)$	$r_1(d)$	$r_1(d)$
s_2	s_2	s_2
$w_3(e)$	$w_2(a)$	$w_2(a)$
$w_2(a)$	$w_1(a)$	$w_1(a)$
$w_1(a)$	$r_1(a)$	$r_1(a)$
$r_1(a)$	$r_1(b)$	$r_1(b)$
$r_1(b)$	$w_3(e)$	$w_3(e)$
s_3	s_3	s_3
$w_1(c)$	$w_1(c)$	$w_1(c)$
S_1	s_1	s_1
$w_1(a)$	$w_1(a)$	$w_3(f)$
$w_3(f)$	$w_3(f)$	$w_1(a)$
$r_2(a)$	$r_2(a)$	$r_2(a)$

Consistency Models – Release Consistency Model

- Are all the propagations necessary?
  - All changes made to the memory by the process are propagated to other nodes.**local update**
  - All changes made to the memory by other processes are propagated from other nodes to the process's node. **remote update**
- Release consistency mode [Gharachorloo et al. 1990] provides a mechanism to clearly tell the system to decide and perform one of these two operations.

similar to semaphore

- Two synchronization variables are required:
  - Acquire: a process is about to enter the critical section.
  - Release: a process is about to leave the critical section.
- Programmers are responsible for putting acquire and release at suitable places in their programs.

Consistency Models – Release Consistency Model

- Requirements for release consistency model:
  - All accesses to *acquire* and *release* synchronization variables obey processor consistency semantics.
  - All previous *acquires* performed by a process must be completed successfully before the process is allowed to perform a data access operation on the memory.
  - All previous data access operations performed by a process must be completed successfully before a *release* access done by the process is allowed.

Consistency Models – Release Consistency Model

Node N_1	Node N_2	Node N_3

Node N_1
```
{
  …
  a1
  a=d
  c=a+b
  r1
  a1
  a=4
  r1
  …
}
```

Node N_2
```
{
  …
  a1
  a=10
  r1
  …
  print(a)
}
```

Node N_3
```
{
  …
  a2
e=foo()
  r2
  a2
  f=bar()
  r2
  …
}
```

Node N_1	Node N_2	Node N_3
…		
$a1_1$	$a1_2$	…
$r_1(d)$	$w_2(a)$ …	
…		
$w_1(a)$		
$r_1(a)$		$a2_3$
$r_1(b)$		$w_3(e)$
…		$r2_3$
$w_1(c)$		$a2_3$
$r1_1$	$r1_2$	
$a1_1$	…	…
$w_1(a)$	$r_2(a)$	
$r1_1$		$w_3(f)$
…		$r2_3$
		…

Global clock

a/r的排列要一樣

Release Consistency Model

Node N_1	Node N_2	Node N_3
$a1_1$	$a1_1$	$a1_1$
$r_1(d)$	$r_1(d)$	$r_1(d)$
$a2_3$	$a2_3$	$a2_3$
$w_3(e)$	$w_3(e)$	$w_3(e)$
$r2_3$	$r2_3$	$r2_3$
$w_1(a)$	$w_1(a)$	$a2_3$
$r_1(a)$	$r_1(a)$	$w_3(f)$
$a2_3$	$r_1(b)$	$r2_3$
$w_3(f)$	$w_1(c)$	$w_1(a)$
$r2_3$	$r1_1$	$r_1(a)$
$r_1(b)$	$a1_2$	$r_1(b)$
$w_1(c)$	$w_2(a)$	$w_1(c)$
$r1_1$	$r1_2$	$r1_1$
$a1_2$	$a2_3$	$a1_2$
$w_2(a)$	$w_3(f)$	$w_2(a)$
$r1_2$	$r2_3$	$r1_2$
$a1_1$	$a1_1$	$a1_1$
$w_1(a)$	$w_1(a)$	$w_1(a)$
$r1_1$	$r1_1$	$r1_1$
$r_2(a)$	$r_2(a)$	$r_2(a)$

Discussion on Consistency Model

- Which model is most intuitive to you?

  strict

- Which model is almost not possible to implement? strict

- Which model is most intuitive to parallel programming model? sequential

- What are the trade-off for weaker consistent model? 僅適用特定情境

Facebook.com

- Suppose facebook.com uses a distributed shared-memory to implement the wall comment/display. Which consistency model should be used so as to minimize the implementation and run-time overhead?

processor consistency

Google Doc

- Suppose that you share your google documents with several groups of friends.
- Which consistency model should be used?

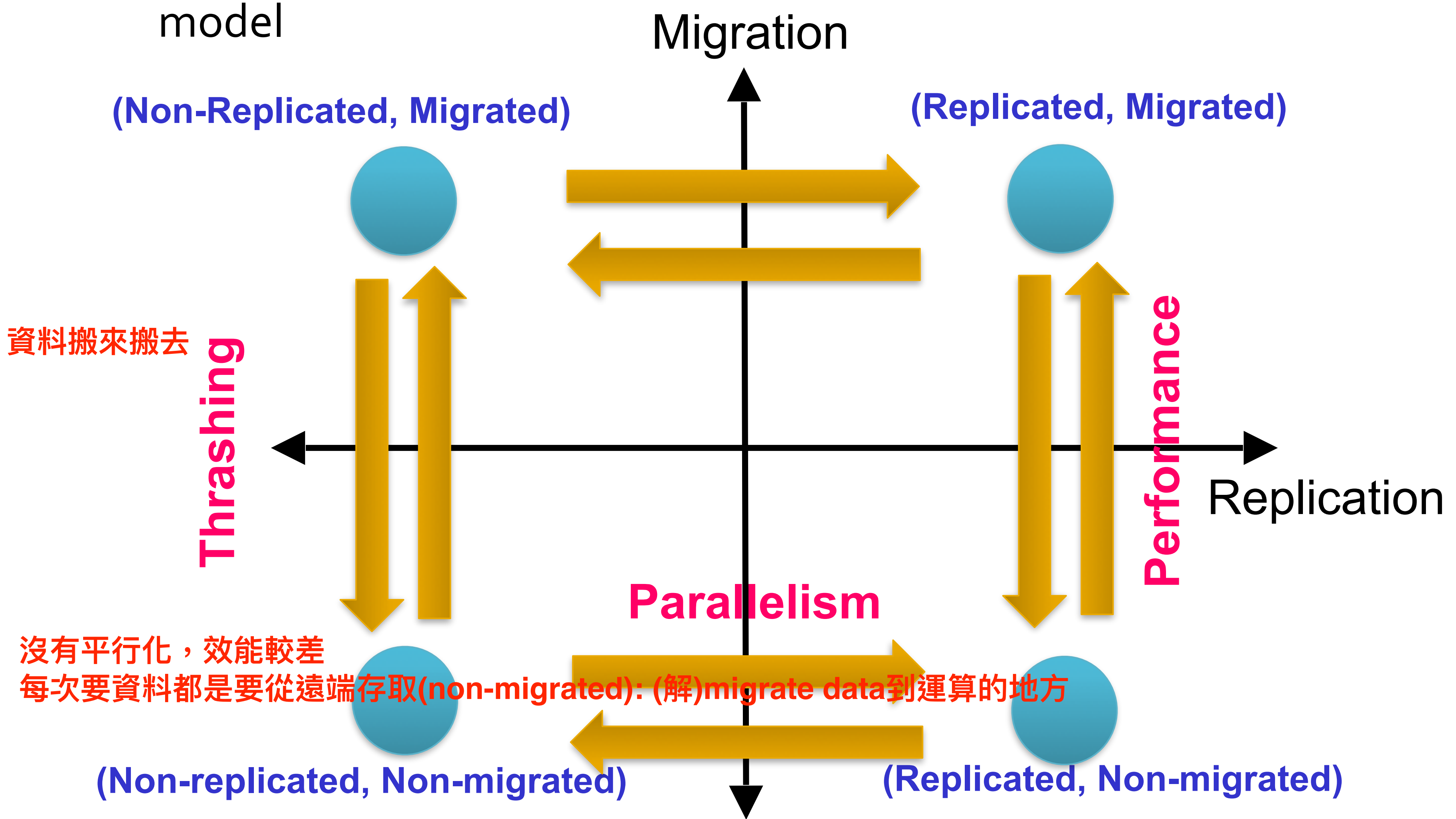
PRAM consistency 沒辦法在同一個位置寫入

  - One document can be edited by at most one user.
  - One document can be edited by more than one user.
    - 'Save' button is required to store data. weak consistency
    - No 'Save' button is required to store data.

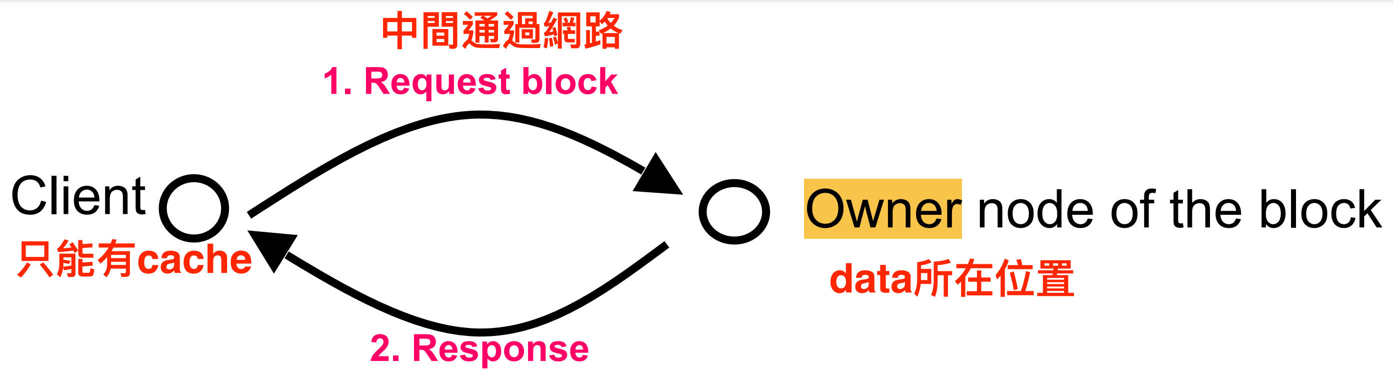
電子白板可能會需要用到sequential

Implement Sequential Consistency Model

- Not practical to implement strict DSM model.
- Replication and migration strategies for sequential consistency model


Migration
(Non-Replicated, Migrated)
(Replicated, Migrated)
資料搬來搬去
Thrashing
Performance
Replication
Parallelism
沒有平行化，效能較差
每次要資料都是要從遠端存取(non-migrated): (解)migrate data到運算的地方
(Non-replicated, Non-migrated)
(Replicated, Non-migrated)

Non-replicated and Non-Migrating Blocks (NRNMB)

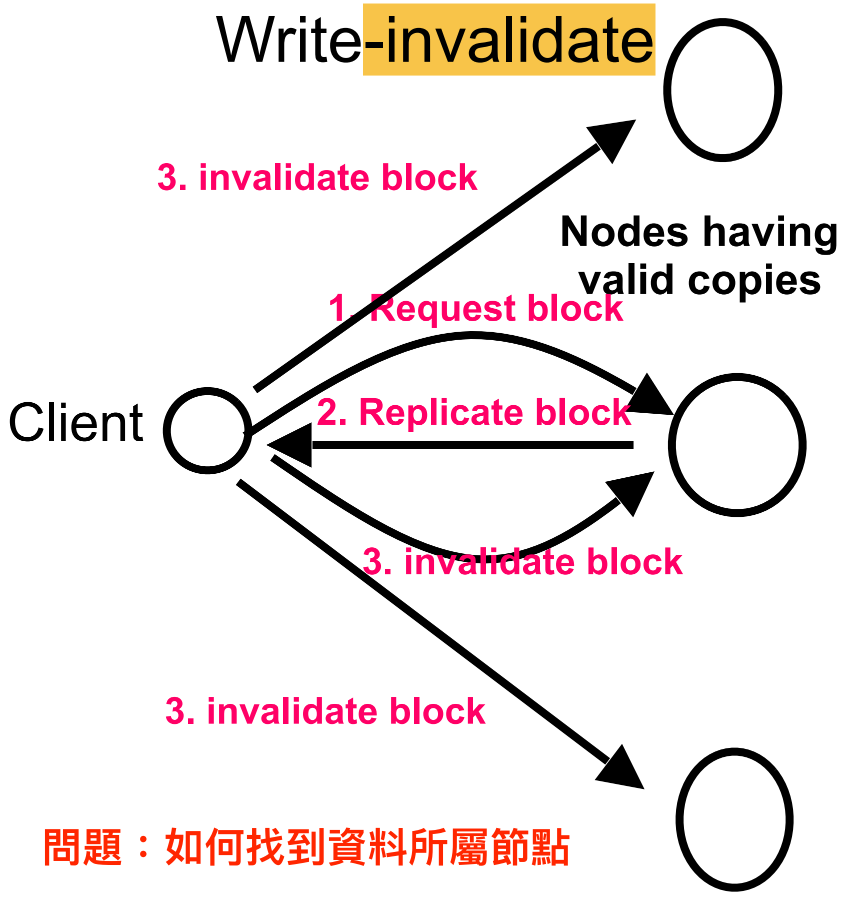


- NRNMB strategy:
  - There is a single copy of each block in the entire system.
  - The location of a block never changes.
- NRNMB is easy to implement but has poor performance when the network latency is high.

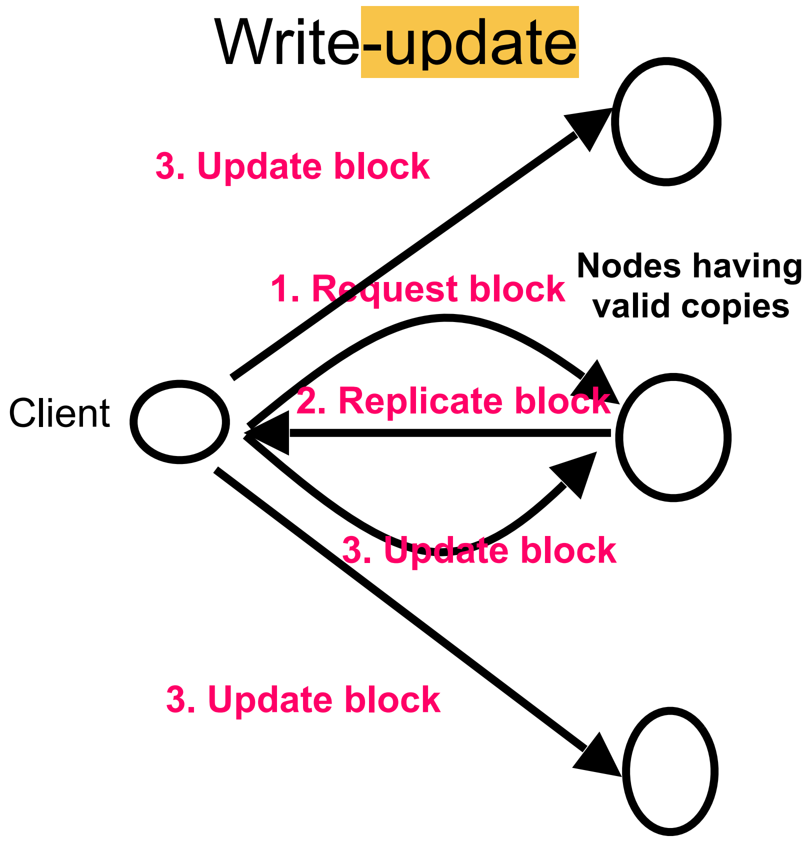
Replicated and migrated blocks

- Replication complicates the memory coherence protocol.
- Two protocols to ensure sequential consistency.

優點：data locality高適合
缺點：容易出現thrashing



問題：如何找到資料所屬節點

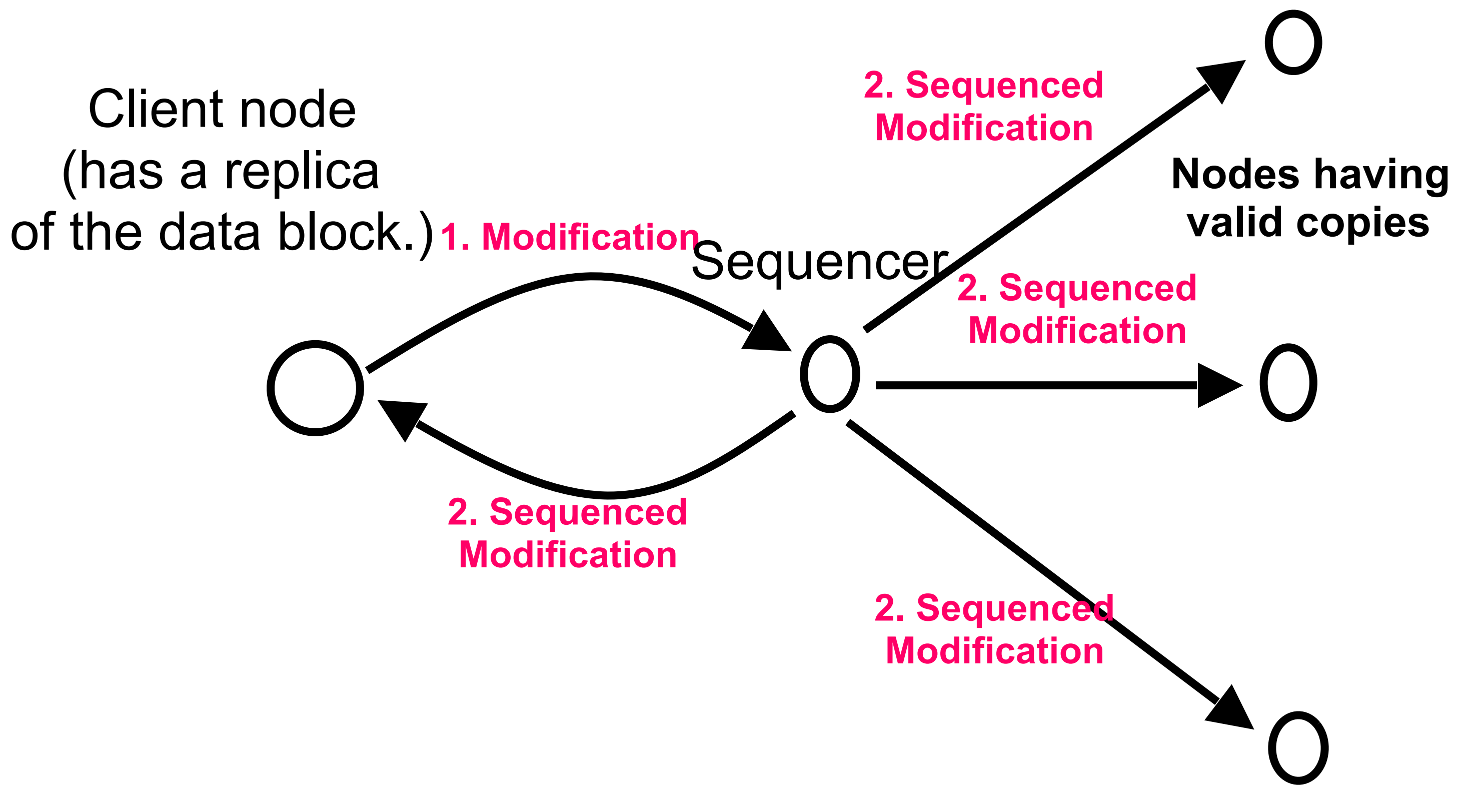


Status Tags for Write-Invalidate Strategy

- The tag indicates
  - whether the block is valid,
  - whether the block is shared, and owner
  - whether the block is read-only or writeable.
- Read Request no ownership transfer
  - If the block is locally available and is valid, the request is satisfied by accessing the local copy.
  - Otherwise, the fault handler generates a read fault and obtains a copy from other nodes.
- Write Request
  - If the block is locally available and is valid and writable, the request is satisfied by accessing the local copy.
  - Otherwise, a fault is generated to obtain a valid copy of the block and ownership change changes its status to writable. The fault also invalidates all other copies of the block. Then, the request can be continued.

Global Sequencing Mechanism

概念與號碼牌機相同


Client node
(has a replica
of the data block.)
1. Modification
Sequencer
2. Sequenced Modification
Nodes having valid copies
2. Sequenced Modification
2. Sequenced Modification
2. Sequenced Modification

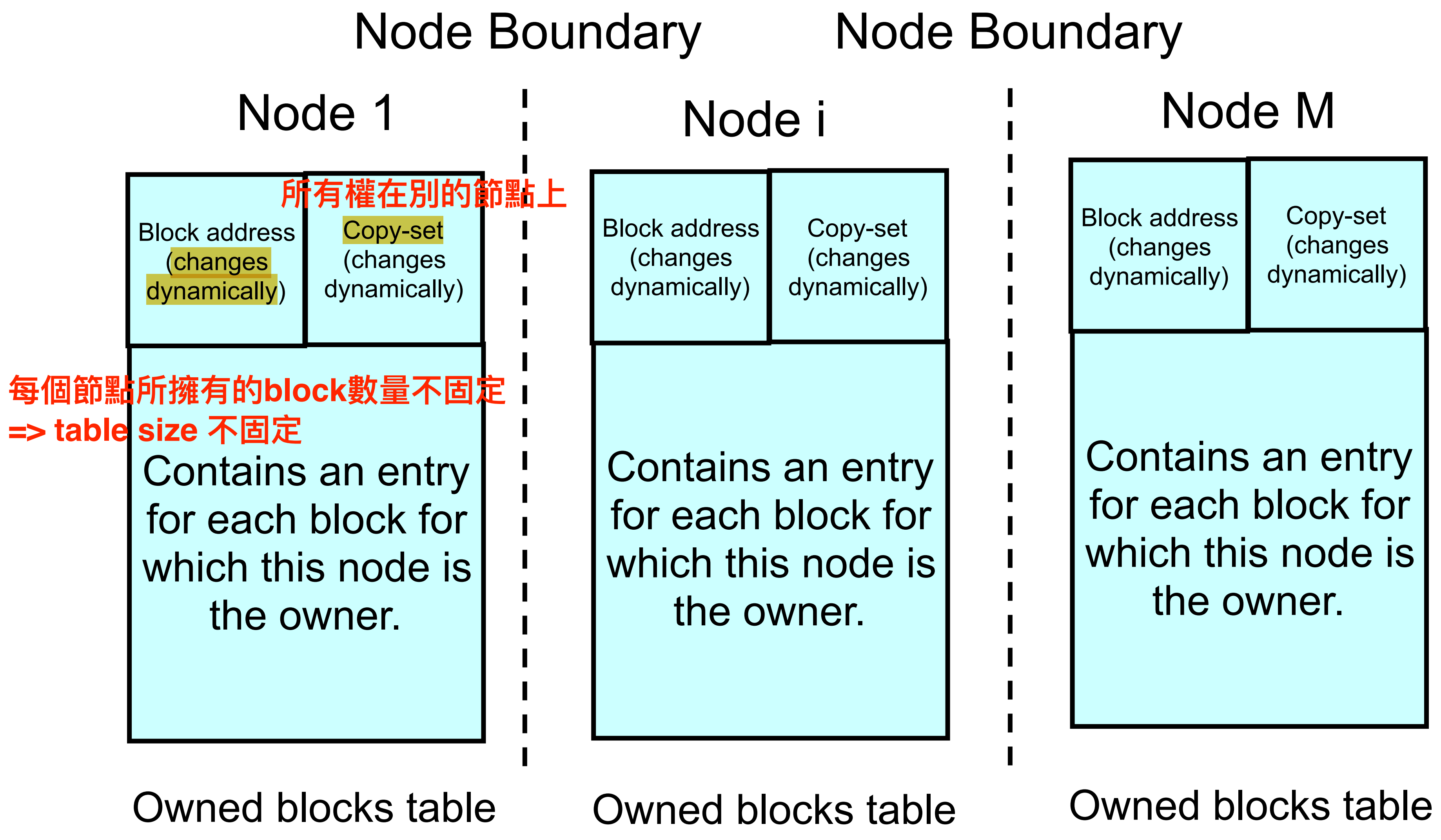
- How to assure that the write operations are totally ordered on every node?
- Write-update is very expensive for use with loosely coupled distributed-memory systems.

Data Locating in the RMB Strategy

- Data locating issues:
  - Locating the owner of a block.
  - Keeping track of the nodes that currently have a valid copy of the block.
- Possible solutions:
  - Broadcasting also centralized algorithm
  - Centralized-server algorithm
  - Fixed distributed-server algorithm
  - Dynamic distributed-server algorithm

Broadcasting Data Locating Mechanism for RMB Strategy

不同block table list 不會重複出現同一個block

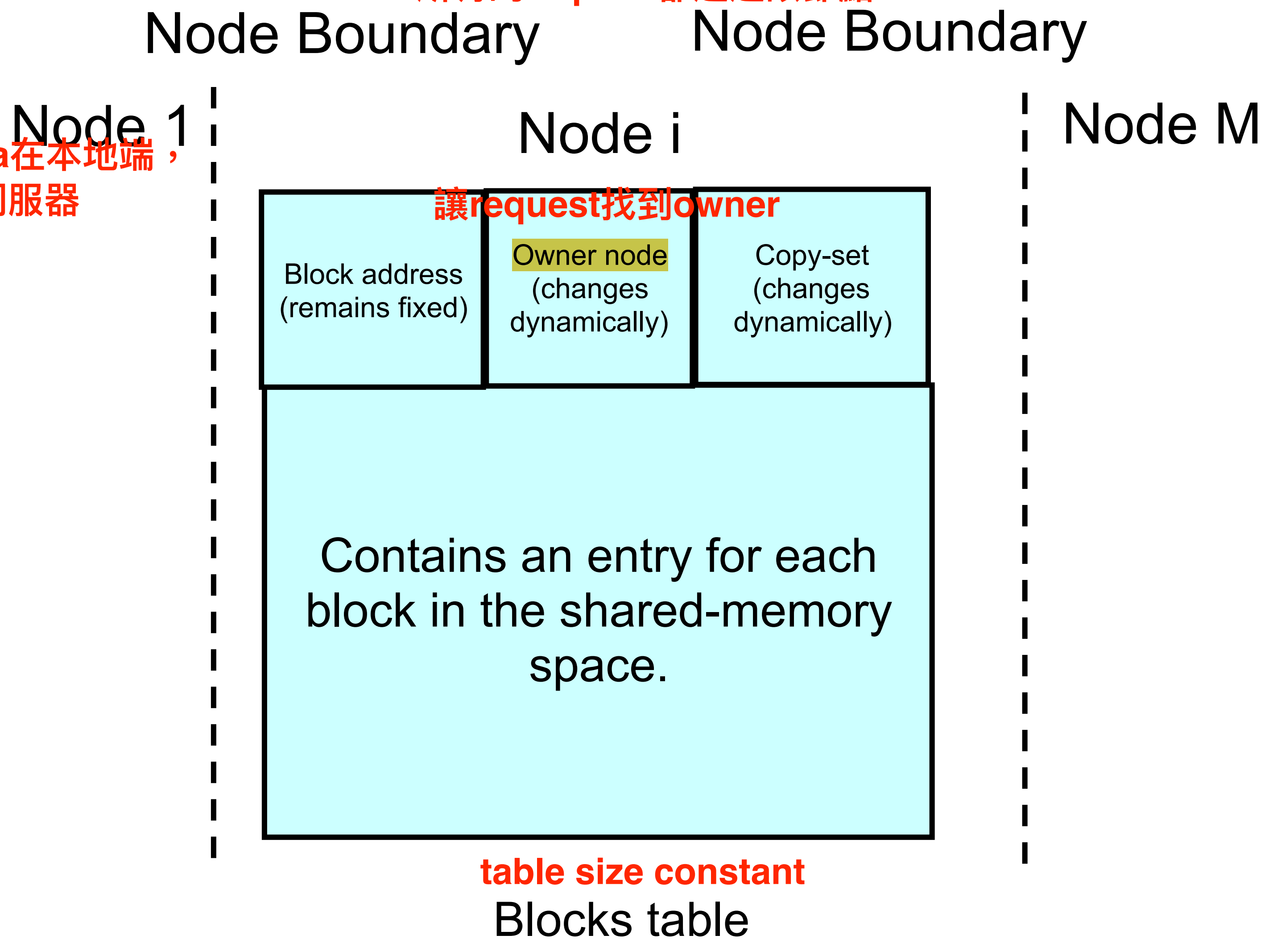


Node Boundary
Node Boundary
Node 1
Node i
Node M
所有權在別的節點上
Block address (changes dynamically)
Copy-set (changes dynamically)
每個節點所擁有的block數量不固定
=> table size 不固定
Contains an entry for each block for which this node is the owner.
Owned blocks table

Centralized-Server Data Locating Mechanism for RMB Strategy

只有一個節點有完整的block table
所有的request都通過該節點

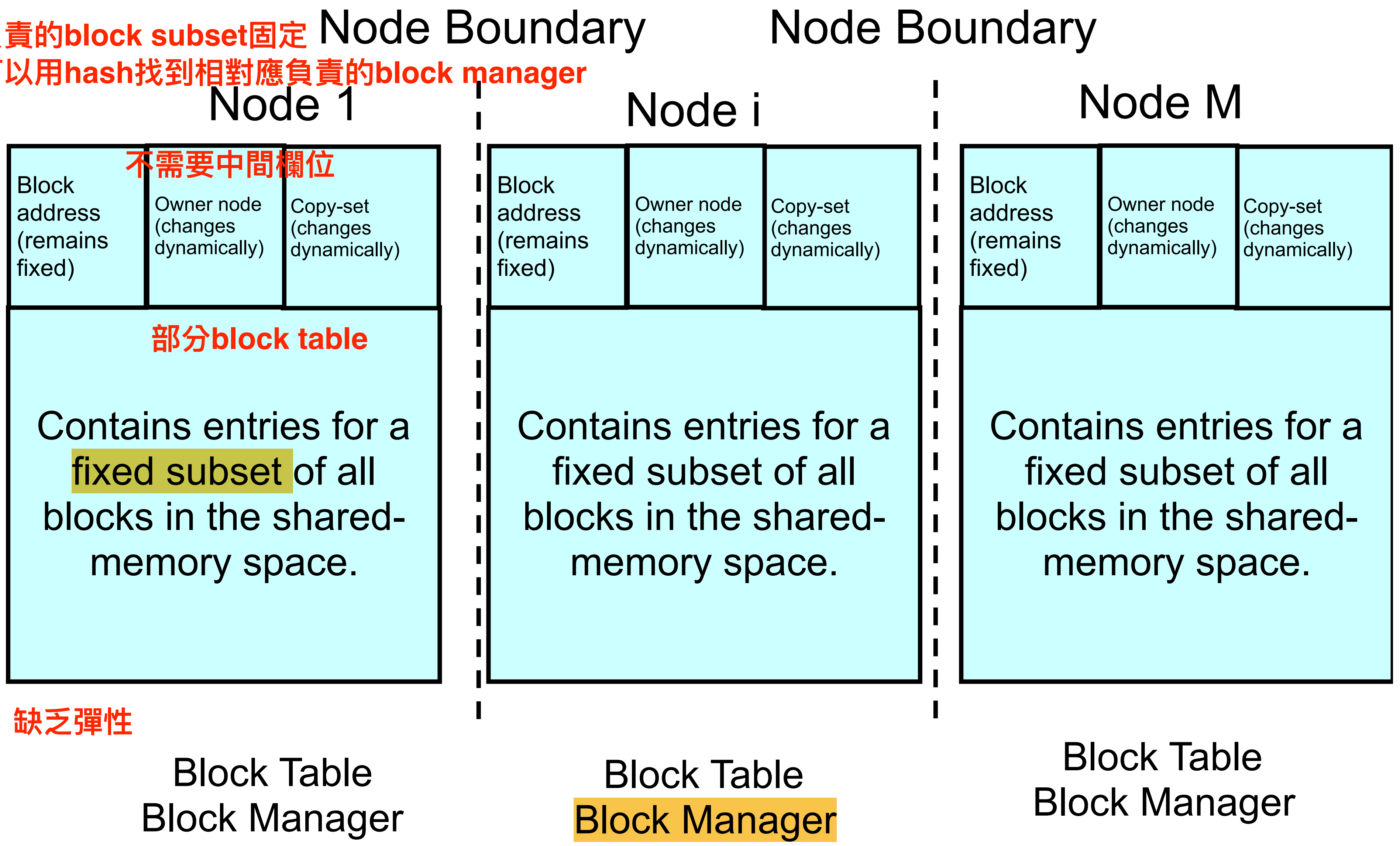
缺點：就算data在本地端，
還是要問中央伺服器


Node Boundary
Node Boundary
Node 1
Node i
Node M
讓request找到owner
Block address (remains fixed)
Owner node (changes dynamically)
Copy-set (changes dynamically)
Contains an entry for each block in the shared-memory space.
table size constant
Blocks table

Distributed-Server Data Locating Mechanism for RMB Strategy

distributed algorithm分散式計算：根據部分資料即可執行任務(ex.不需要global clock)

負責的block subset固定
可以用hash找到相對應負責的block manager



Node Boundary

Node Boundary

Node 1

Block address (remains fixed)	Owner node (changes dynamically)	Copy-set (changes dynamically)

不需要中間欄位

部分block table

Contains entries for a fixed subset of all blocks in the shared-memory space.

Block Table
Block Manager

Node i

Block address (remains fixed)	Owner node (changes dynamically)	Copy-set (changes dynamically)

Contains entries for a fixed subset of all blocks in the shared-memory space.

Block Table
Block Manager

Node M

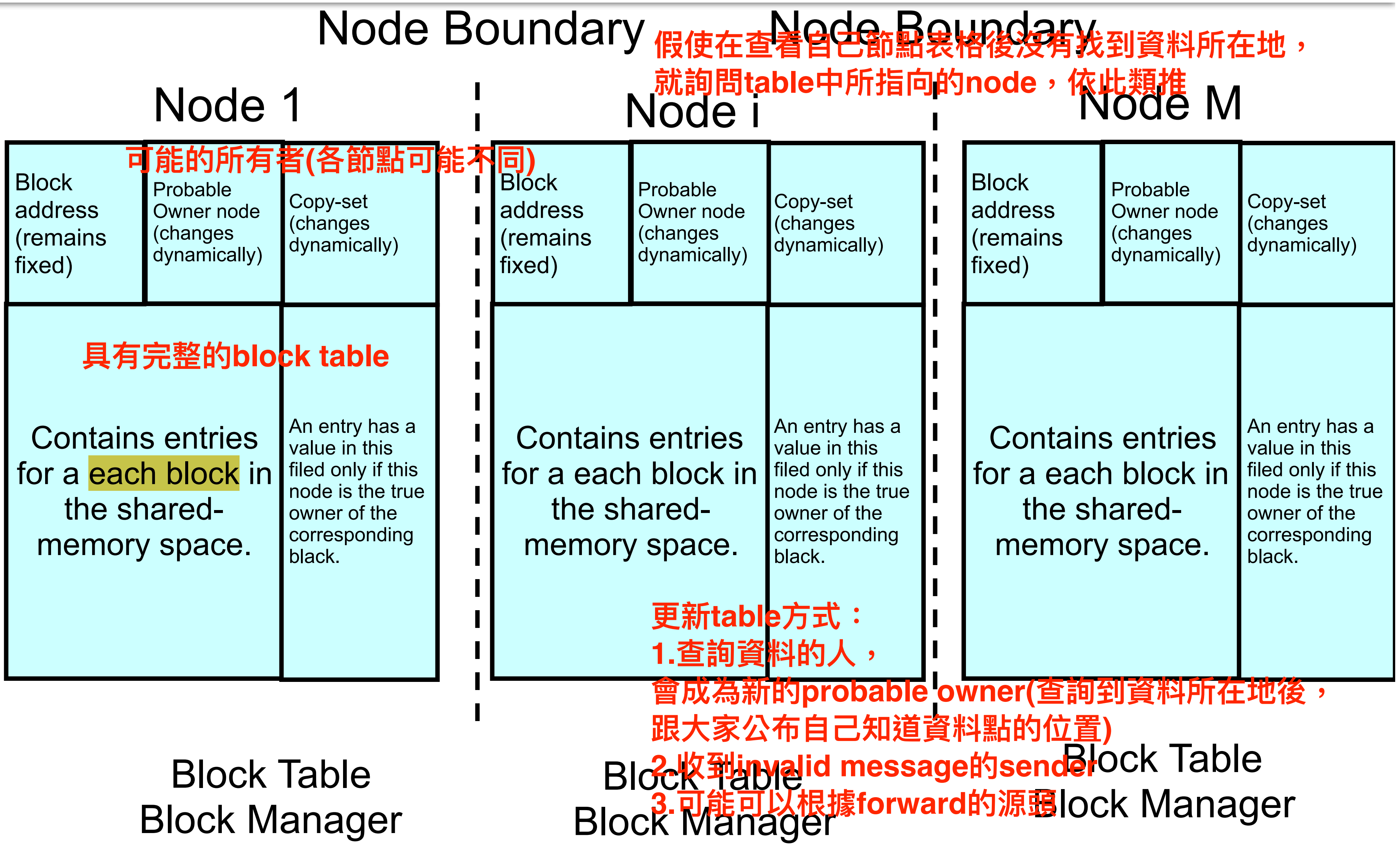
Block address (remains fixed)	Owner node (changes dynamically)	Copy-set (changes dynamically)

Contains entries for a fixed subset of all blocks in the shared-memory space.

Block Table
Block Manager

缺點：缺乏彈性

Dynamic Distributed-Server Data Locating Mechanism for RMB Strategy



假使在查看自己節點表格後沒有找到資料所在地，
就詢問table中所指向的node，依此類推

可能的所有者(各節點可能不同)

Block address (remains fixed)	Probable Owner node (changes dynamically)	Copy-set (changes dynamically)
Contains entries for a each block in the shared-memory space.		An entry has a value in this filed only if this node is the true owner of the corresponding black.

具有完整的block table

更新table方式：
1.查詢資料的人，
會成為新的probable owner(查詢到資料所在地後，
跟大家公布自己知道資料點的位置)
2.收到invalid message的sender
3.可能可以根據forward的源頭

Replacement Strategy

- Challenging Issues for caching shared data:
  - Which block to replace?
  - Where to place a replaced block?
- Replacement Algorithms:
  - Usage-based versus non-usage based: LRU vs. FIFO
  - Fixed space versus variable space
    - Is variable space suitable?

DSM in IVY [Li 1986, 1988]

- Most DSM differentiate the status of data items and use a priority mechanism.
- Each memory block is classified into one of the following five types: unused, nil, read-only, read-owned, and writable.
- Replacement Priority:
  - Both unused and nil have the highest replacement priority. (Note: LRU may leave nil blocks as they are invalidated recently.)
  - Read-only blocks are the next.
  - Read-owned and writable blocks for which replica(s) exist on some other node(s) are the next.
  - Read-owned and writable blocks for which only this node has

Where to place a replaced block

- Two commonly used approaches:
  - Using secondary storage
  - Using the memory space of other nodes.

Thrashing

- Why thrashing?
  - Data blocks are moved back and forth in quick succession.
  - Blocks with read-only permissions are repeatedly invalidated soon after they are replicated.
- Thrashing indicates poor (node) locality in references.
- Avoid thrashing:
  - Providing application-controlled locks
  - Nailing a block to a node for a minimum amount of time
  - Tailoring the coherence algorithms to the shared-data usage patterns

Heterogeneous DSM

Two main issues:

- Data Conversion: it is necessary to take assistance from application programmers.
  - Structuring the DSM system as a collection of source language objects.
  - Allowing only one type of data in a block.
- Selection of block size
  - Largest page size algorithm
  - Smallest page size algorithm
  - Intermediate page size algorithm