

# *Distributed Systems*

## *Consistency and Replication*

Alberto Montresor  
Università di Trento

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

This work is partially based on slides and pictures of Marteen van Steen, Lorenzo Alvisi.

# Availability

- ◆ **Def: The probability that a system will provide its required service**
- ◆ **Example**
  - ◆ one single server
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
- ◆ **Example**
  - ◆ 30 servers
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
  - ◆ all required to perform the service

# Availability

- ◆ **Def: The probability that a system will provide its required service**
- ◆ **Example**
  - ◆ one single server
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
- ◆ **Example**
  - ◆ 30 servers
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
  - ◆ all required to perform the service
- ◆ **Availability**

# Availability

- ◆ **Def: The probability that a system will provide its required service**
- ◆ **Example**
  - ◆ one single server
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
- ◆ **Example**
  - ◆ 30 servers
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
  - ◆ all required to perform the service
- ◆ **Availability**
  - ◆ 1 week = 10080 minutes

# Availability

- ◆ **Def:** The probability that a system will provide its required service
- ◆ **Example**
  - ◆ one single server
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
- ◆ **Example**
  - ◆ 30 servers
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
  - ◆ all required to perform the service
- ◆ **Availability**
  - ◆ 1 week = 10080 minutes
  - ◆ availability =  $1 - p_{crash}$  =

# Availability

- ◆ **Def:** The probability that a system will provide its required service
- ◆ **Example**
  - ◆ one single server
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
- ◆ **Example**
  - ◆ 30 servers
  - ◆ crash once per week
  - ◆ takes 2 minutes to reboot
  - ◆ all required to perform the service
- ◆ **Availability**
  - ◆ 1 week = 10080 minutes
  - ◆ availability =  $1 - p_{crash} = 1 - 2 * 10^{-4} = 0.9998$

# Availability

- ◆ **Def:** The probability that a system will provide its required service

- ◆ **Example**

- ◆ one single server
- ◆ crash once per week
- ◆ takes 2 minutes to reboot

- ◆ **Example**

- ◆ 30 servers
- ◆ crash once per week
- ◆ takes 2 minutes to reboot
- ◆ all required to perform the service

- ◆ **Availability**

- ◆ 1 week = 10080 minutes
- ◆ availability =  $1 - p_{crash} =$   
 $1 - 2 * 10^{-4} = 0.9998$

- ◆ **Availability**

# Availability

- ◆ **Def:** The probability that a system will provide its required service

- ◆ **Example**

- ◆ one single server
- ◆ crash once per week
- ◆ takes 2 minutes to reboot

- ◆ **Example**

- ◆ 30 servers
- ◆ crash once per week
- ◆ takes 2 minutes to reboot
- ◆ all required to perform the service

- ◆ **Availability**

- ◆ 1 week = 10080 minutes
- ◆ availability =  $1 - p_{crash} =$   
 $1 - 2 * 10^{-4} = 0.9998$

- ◆ **Availability**

- ◆ availability =  $(1 - p_{crash})^{30} =$

# Availability

- ◆ **Def:** The probability that a system will provide its required service

- ◆ **Example**

- ◆ one single server
- ◆ crash once per week
- ◆ takes 2 minutes to reboot

- ◆ **Example**

- ◆ 30 servers
- ◆ crash once per week
- ◆ takes 2 minutes to reboot
- ◆ all required to perform the service

- ◆ **Availability**

- ◆ 1 week = 10080 minutes
- ◆ availability =  $1 - p_{crash} =$   
 $1 - 2 * 10^{-4} = 0.9998$

- ◆ **Availability**

- ◆ availability =  $(1 - p_{crash})^{30} =$   
0.994

# Availability

- ◆ **Def:** The probability that a system will provide its required service

- ◆ **Example**

- ◆ one single server
- ◆ crash once per week
- ◆ takes 2 minutes to reboot

- ◆ **Example**

- ◆ 30 servers
- ◆ crash once per week
- ◆ takes 2 minutes to reboot
- ◆ all required to perform the service

- ◆ **Availability**

- ◆ 1 week = 10080 minutes
- ◆ availability =  $1 - p_{crash} =$   
 $1 - 2 \cdot 10^{-4} = 0.9998$

- ◆ **Availability**

- ◆ availability =  $(1 - p_{crash})^{30} =$   
0.994

- ◆ **Five-nines**

# Availability

- ◆ **Def:** The probability that a system will provide its required service

- ◆ **Example**

- ◆ one single server
- ◆ crash once per week
- ◆ takes 2 minutes to reboot

- ◆ **Example**

- ◆ 30 servers
- ◆ crash once per week
- ◆ takes 2 minutes to reboot
- ◆ all required to perform the service

- ◆ **Availability**

- ◆ 1 week = 10080 minutes
- ◆ availability =  $1 - p_{crash} =$   
 $1 - 2 \cdot 10^{-4} = 0.9998$

- ◆ **Availability**

- ◆ availability =  $(1 - p_{crash})^{30} =$   
0.994

- ◆ **Five-nines**

- ◆ 0.99999 = 5 minutes x year

# Availability

- ◆ **How to increase availability**

- ◆ Avoid single point of failures
- ◆ Use replication (time/space)

- ◆ **Replication in space**

- ◆ Run parallel copies
- ◆ Vote on replica output
- ◆ High-availability, high-cost

- ◆ **Replication in time**

- ◆ When a replica fails, restart it (or replace it)
- ◆ Lower maintenance, lower availability

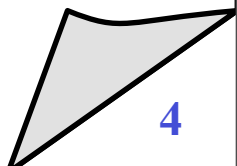
- ◆ **Example**

- ◆ Replicate server 30 times

- ◆ **Availability**

- ◆  $\text{availability} = 1 - p_{\text{crash}}^{30} =$   
 $\sim 1 - 10^{111}$

- ◆ **Availability:**
  - ◆ Replicating a service increases its availability
- ◆ **Performance**
  - ◆ Geographical location
  - ◆ Load-balancing
  - ◆ No bottlenecks
- ◆ **But:**
  - ◆ Replication needs reliability and consistency
  - ◆ Trade-off between consistency and scalability



# Consistency

- ◆ **Multiple copies may lead to consistency problems.**
  - ◆ Whenever a copy is modified, that copy becomes different from the rest.
  - ◆ Modifications have to be carried out on all copies to ensure consistency.
  - ◆ The type of application has an impact on the consistency requirements needed and thus on the implementation.

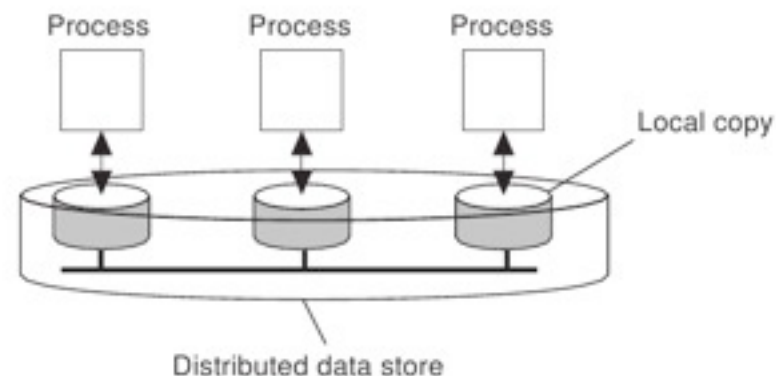
# What is the problem

- ◆ **Main goal:**
  - ◆ To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the the same order everywhere
- ◆ **Conflicting operations - from the world of transactions:**
  - ◆ *Read–write conflict*: concurrent read operation and write operation
  - ◆ *Write–write conflict*: two concurrent write operations
- ◆ **Problem**
  - ◆ Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability
- ◆ **Solution:**
  - ◆ Weaken consistency requirements so that hopefully global synchronization can be avoided

# Consistency model

## ◆ Definitions

- ◆ Consistency model
  - ◆ A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.
- ◆ Data store
  - ◆ a distributed collection of storages accessible to clients



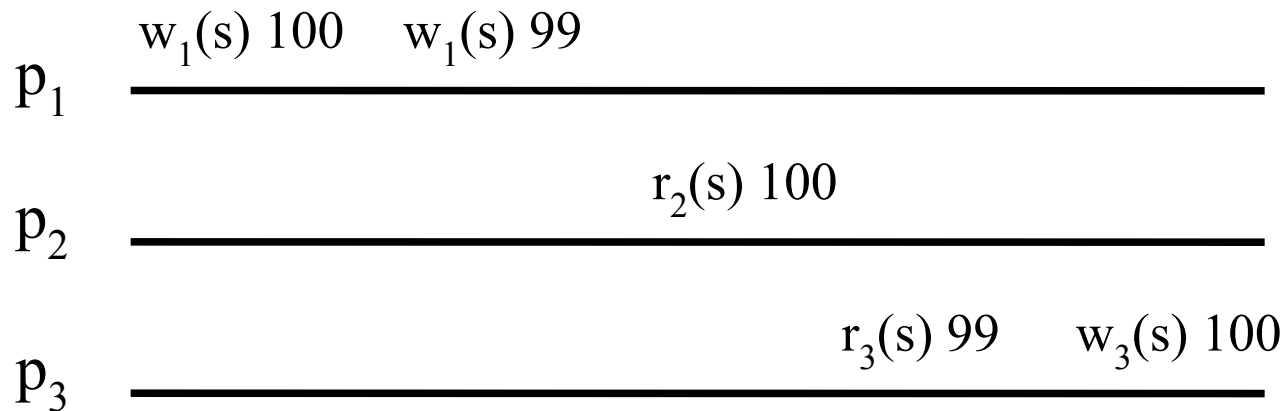
# Consistency models

- ◆ **Data-centric consistency models**
  - ◆ Strict consistency
  - ◆ Linearizability
  - ◆ Sequential consistency
  - ◆ Causal consistency / FIFO consistency
- ◆ **Client-centric consistency models**
  - ◆ Eventual consistency
  - ◆ Consistency models for mobile clients
    - ◆ Read-after-read (monotonic read)
    - ◆ Write-after-write (monotonic write)
    - ◆ Read-after-write (read your writes)
    - ◆ Write-after-read (write follows read)

- ◆ **Flight reservation database**
  - ◆ At 9.36, all seats of flight 48 are booked
  - ◆ At 9.37, user *C* cancel its reservation on flight 48
  - ◆ At 9.38, user *A* tries to reserve a seat on flight 48 – the answer is fully booked
  - ◆ At 9.39, user *B* tries to reserve a set on flight 48 – the seat is granted
- ◆ **What do you think?**

# Notation

- ◆ **Write operation:  $w_i(x)a$** 
  - ◆ Proc.  $p_i$  has written  $a$  on variable  $x$
- ◆ **Read operation:  $r_i(x)a$** 
  - ◆ Proc.  $p_i$  has read  $a$  from variable  $x$



- ◆ **Definition**

- ◆ A read operation has to return the result of the latest write operation which occurred on the data item.

- ◆ **Implementation**

- ◆ Only possible with a global, perfectly synchronized clock
- ◆ Only possible if all writes instantaneously visible to all

- ◆ **It makes sense, though:**

- ◆ it is the model of uniprocessor systems!

# Linearizability – Herlihy and Wing, 1991

## ◆ Definition

- 1) The result of any execution is the same as if the operations by all procs on the data store were executed in some sequential order
- 2) The operation of each process appear in this sequence in the order specified by its program
- 3) If  $t_1$  and  $t_2$  are the times at which two distinct processes perform operations  $o_1$  and  $o_2$ , and  $t_1 < t_2$ , then  $o_1$  must appear before  $o_2$  in the sequence

## ◆ Remember

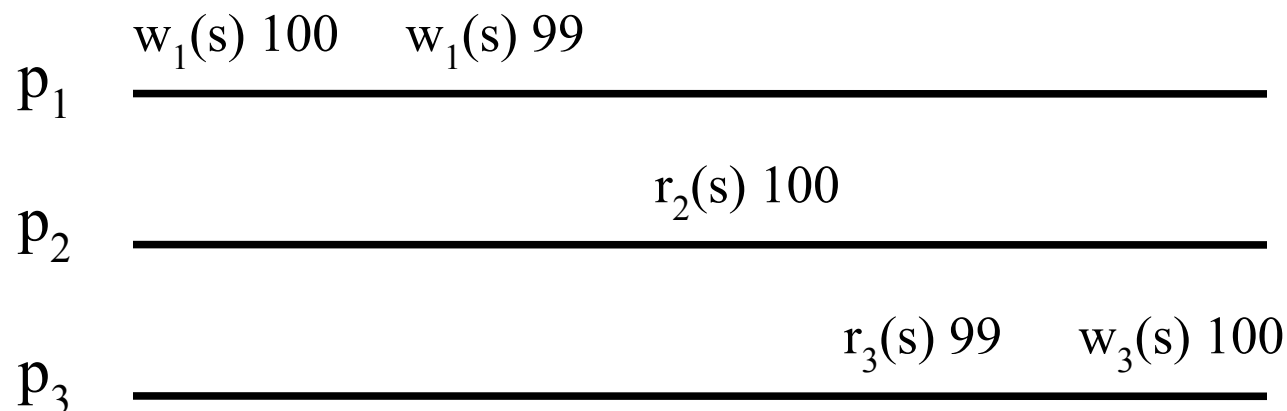
- ◆ The sequence of events corresponds to a run

◆ **Is the example below linearizable? (\*)**

◆  $w_1(s)100 - w_1(s)99 - r_2(s)100 - r_3(s)99 - w_3(s)100$

◆  $w_1(s)100 - r_2(s)100 - w_1(s)99 - r_3(s)99 - w_3(s)100$

◆ **(\*) Read: assume a replication protocol that produces this sequence of actions; is this replication protocol linearizable?**



# Sequential consistency – Lamport, 1978

## ◆ Definition

- ◆ The result of any execution is the same as if the operations by all procs on the data store were executed in some sequential order
- ◆ The operation of each process appear in this sequence in the order specified by its program
- ~~◆ If  $t_1$  and  $t_2$  are the times at which two distinct processes perform operations  $o_1$  and  $o_2$ , and  $t_1 < t_2$ , then  $o_1$  must appear before  $o_2$  in the sequence~~

## ◆ Comments

- ◆ Much more common

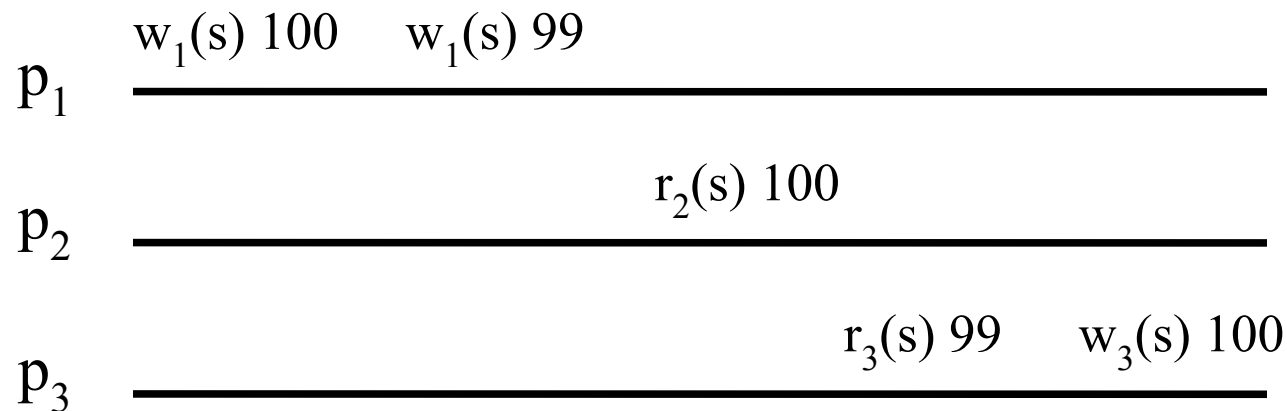
# Sequential consistency – Lamport, 1978

\*\*

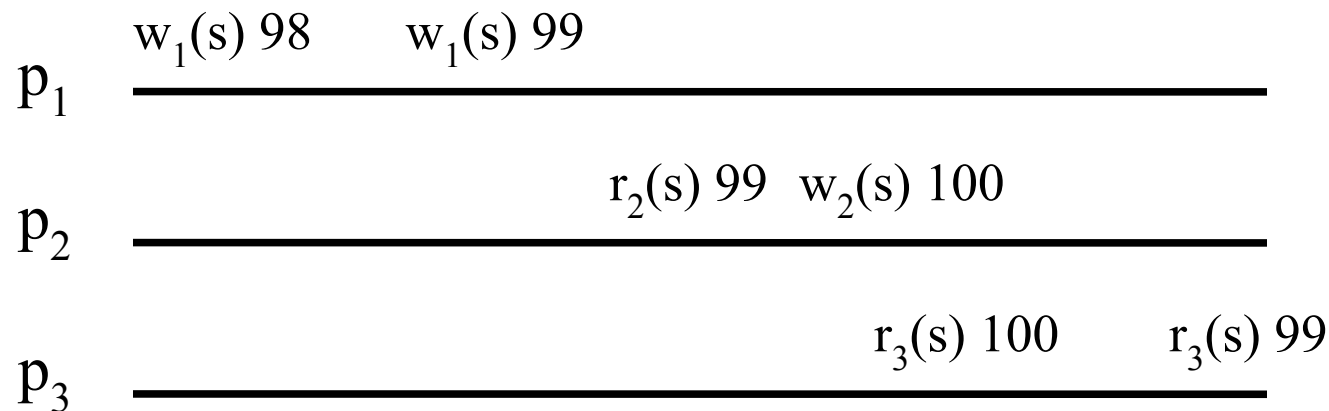
- ◆ Is the example below sequentially consistent? (\*)

- ◆  $w_1(s)100 - r_2(s)100 - w_1(s)99 - r_3(s)99 - w_3(s)100$

- ◆ (\*) Read: assume a replication protocol that produces this sequence of actions; is this replication protocol sequentially consistent?



- ◆ Is the example below sequentially consistent? (\*)
  - ◆ From 1,2:  $w_1(s)99 - r_2(s)99 - w_2(s)100$
  - ◆ From 3:  $r_3(s) 100 - r_3(s)99$
- ◆ (\*) Read: assume a replication protocol that produces this sequence of actions; is this replication protocol sequentially consistent?



# Sequential consistency: example

\*\*

Process $P_1$	Process $P_2$	Process $P_3$
$x := 1$	$y := 1$	$z := 1$
print $y, z$	print $x, z$	print $x, y$

- ◆ **How many “potential executions”?**
  - ◆ Executions without requiring conditions 1,2
- ◆ **How many “valid executions”?**
  - ◆ Executions requiring condition 2
- ◆ **How many “potential outputs”?**
  - ◆ Signatures, considering  $P_1, P_2, P_3$  in order

# Sequential consistency: example

\*\*

## ◆ Process $P_1$

$x := 1$

print  $y, z$

## Process $P_2$

$y := 1$

print  $x, z$

## Process $P_3$

$z := 1$

print  $x, y$

## ◆ How many “potential executions”?

720 (6!)

- ◆ Executions without requiring conditions 1,2

## ◆ How many “valid executions”?

- ◆ Executions requiring condition 2

## ◆ How many “potential outputs”?

- ◆ Signatures, considering  $P_1, P_2, P_3$  in order

# Sequential consistency: example

\*\*

## ◆ Process $P_1$

$x := 1$

print  $y, z$

## Process $P_2$

$y := 1$

print  $x, z$

## Process $P_3$

$z := 1$

print  $x, y$

## ◆ How many “potential executions”?

720 (6!)

- ◆ Executions without requiring conditions 1,2

## ◆ How many “valid executions”?

90  $(5!/4)*3$

- ◆ Executions requiring condition 2

## ◆ How many “potential outputs”?

- ◆ Signatures, considering  $P_1, P_2, P_3$  in order

# Sequential consistency: example

\*\*

## ◆ Process $P_1$

$x := 1$

print y,z

## Process $P_2$

$y := 1$

print x,z

## Process $P_3$

$z := 1$

print x,y

## ◆ How many “potential executions”?

720 (6!)

- ◆ Executions without requiring conditions 1,2

## ◆ How many “valid executions”?

90  $(5!/4)*3$

- ◆ Executions requiring condition 2

## ◆ How many “potential outputs”?

64  $(2^6)$

- ◆ Signatures, considering  $P_1, P_2, P_3$  in order

# Sequential consistency: example

\*\*

◆ **Process  $P_1$**

$x := 1$

print y,z

**Process  $P_2$**

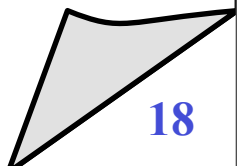
$y := 1$

print x,z

**Process  $P_3$**

$z := 1$

print x,y-axes



# Sequential consistency: example

\*\*

◆ **Process  $P_1$**

$x := 1$

print y,z

**Process  $P_2$**

$y := 1$

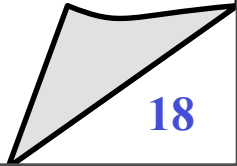
print x,z

**Process  $P_3$**

$z := 1$

print x,y-axes

◆ **How many “seq. consistent outputs”?**      **< 64**



# Sequential consistency: example

\*\*

Process $P_1$	Process $P_2$	Process $P_3$
$x := 1$	$y := 1$	$z := 1$
print $y, z$	print $x, z$	print $x, y$ -axes

- ◆ How many “seq. consistent outputs”?  $< 64$ 
  - ◆ Example: Is 000000 sequentially consistent? Why?

# Sequential consistency: example

\*\*

Process $P_1$	Process $P_2$	Process $P_3$
$x := 1$	$y := 1$	$z := 1$
print y,z	print x,z	print x,y-axes

- ◆ How many “seq. consistent outputs”? < 64
  - ◆ Example: Is 000000 sequentially consistent? Why?
    - ◆ All print operations “happen before” the updates - impossible

# Sequential consistency: example

\*\*

Process $P_1$	Process $P_2$	Process $P_3$
$x := 1$	$y := 1$	$z := 1$
print $y, z$	print $x, z$	print $x, y$ -axes

- ◆ How many “seq. consistent outputs”? < 64
  - ◆ Example: Is 000000 sequentially consistent? Why?
    - ◆ All print operations “happen before” the updates - impossible
  - ◆ Example: Is 001001 sequentially consistent? Why?

# Sequential consistency: example

\*\*

Process $P_1$	Process $P_2$	Process $P_3$
$x := 1$	$y := 1$	$z := 1$
print y,z	print x,z	print x,y-axes

- ◆ How many “seq. consistent outputs”?  $< 64$ 
  - ◆ Example: Is 000000 sequentially consistent? Why?
    - ◆ All print operations “happen before” the updates - impossible
  - ◆ Example: Is 001001 sequentially consistent? Why?
    - ◆ print yz=00 after  $x:=1$ , before  $y:=1$ ,  $z:=1$

# Sequential consistency: example

\*\*

Process $P_1$	Process $P_2$	Process $P_3$
$x := 1$	$y := 1$	$z := 1$
print y,z	print x,z	print x,y-axes

- ◆ How many “seq. consistent outputs”? < 64
  - ◆ Example: Is 000000 sequentially consistent? Why?
    - ◆ All print operations “happen before” the updates - impossible
  - ◆ Example: Is 001001 sequentially consistent? Why?
    - ◆ print yz=00 after  $x:=1$ , before  $y:=1$ ,  $z:=1$ 
      - ◆  $x := 1$  – print yz=00 –  $y:=1$  – print xz=10 –  $z:=1$  – print xy=11

# Sequential consistency: example

\*\*

◆ Process $P_1$	Process $P_2$	Process $P_3$
$x := 1$	$y := 1$	$z := 1$
print y,z	print x,z	print x,y-axes

- ◆ How many “seq. consistent outputs”? < 64
  - ◆ Example: Is 000000 sequentially consistent? Why?
    - ◆ All print operations “happen before” the updates - impossible
  - ◆ Example: Is 001001 sequentially consistent? Why?
    - ◆ print yz=00 after  $x:=1$ , before  $y:=1$ ,  $z:=1$ 
      - ◆  $x := 1$  – print yz=00 –  $y:=1$  – print xz=10 –  $z:=1$  – print xy=11
      - ◆  $x := 1$  – print yz=00 –  $z:=1$  – no (z was not equal to 1)

## ◆ Definition

- ◆ All writes that are (potentially) causally related must be seen by every process in the same order

## ◆ Define “causally related”:

- ◆ a read followed by a write, on the same process:
  - ◆ the write is (potentially) causally related by the read
- ◆ a write followed by a read of the same value, on diff. process:
  - ◆ the read is (potentially) causally related by the write

## ◆ Example of use:

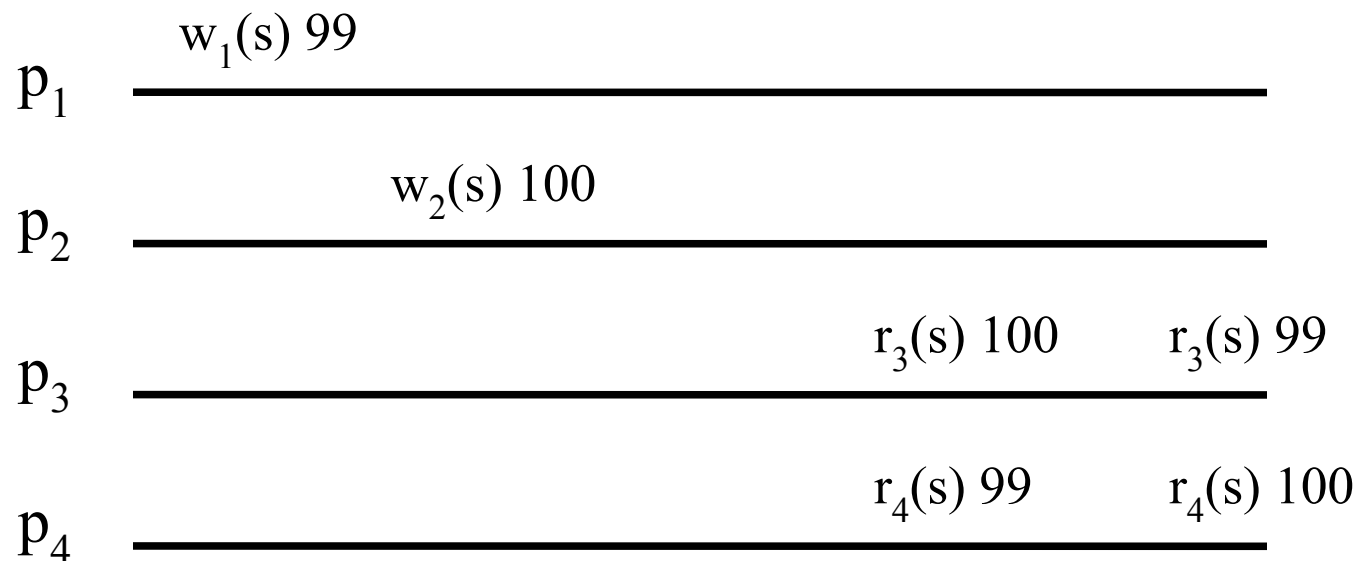
- ◆ Bulletin board

## ◆ Definition

- ◆ All writes that are (potentially) causally related must be seen by every process in the same order

## ◆ Example

- ◆ Is the following example causally consistent?
- ◆ Is the following example sequentially consistent?

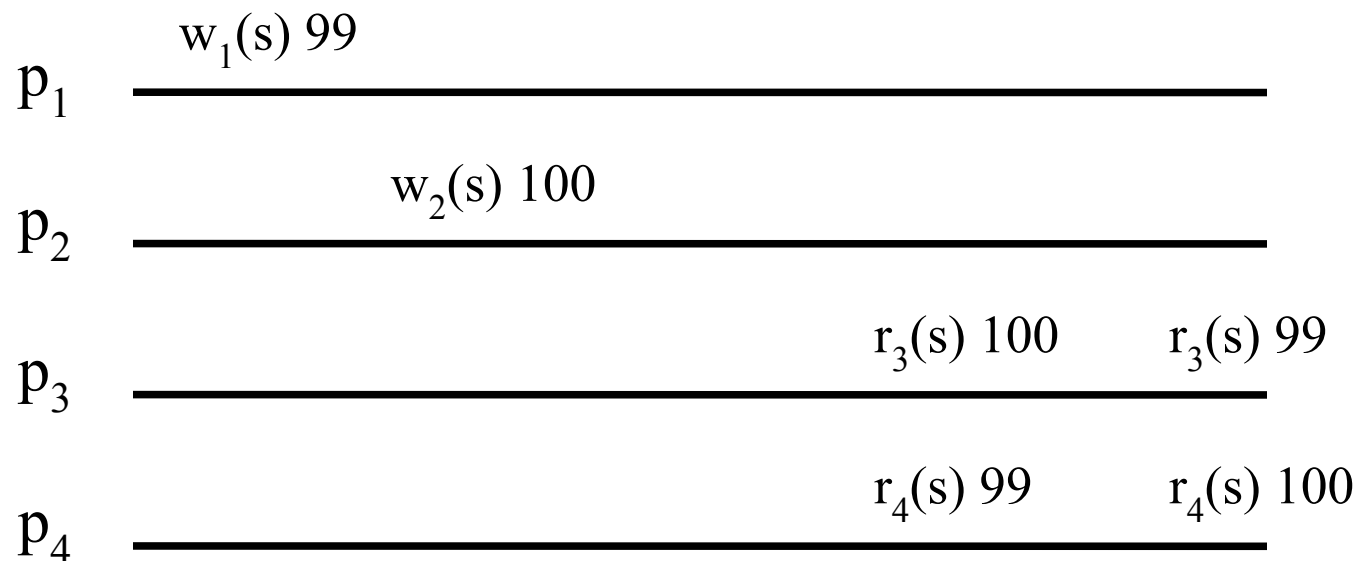


## ◆ Definition

- ◆ All writes that are (potentially) causally related must be seen by every process in the same order

## ◆ Example

- ◆ Is the following example causally consistent? **Yes!**
- ◆ Is the following example sequentially consistent?

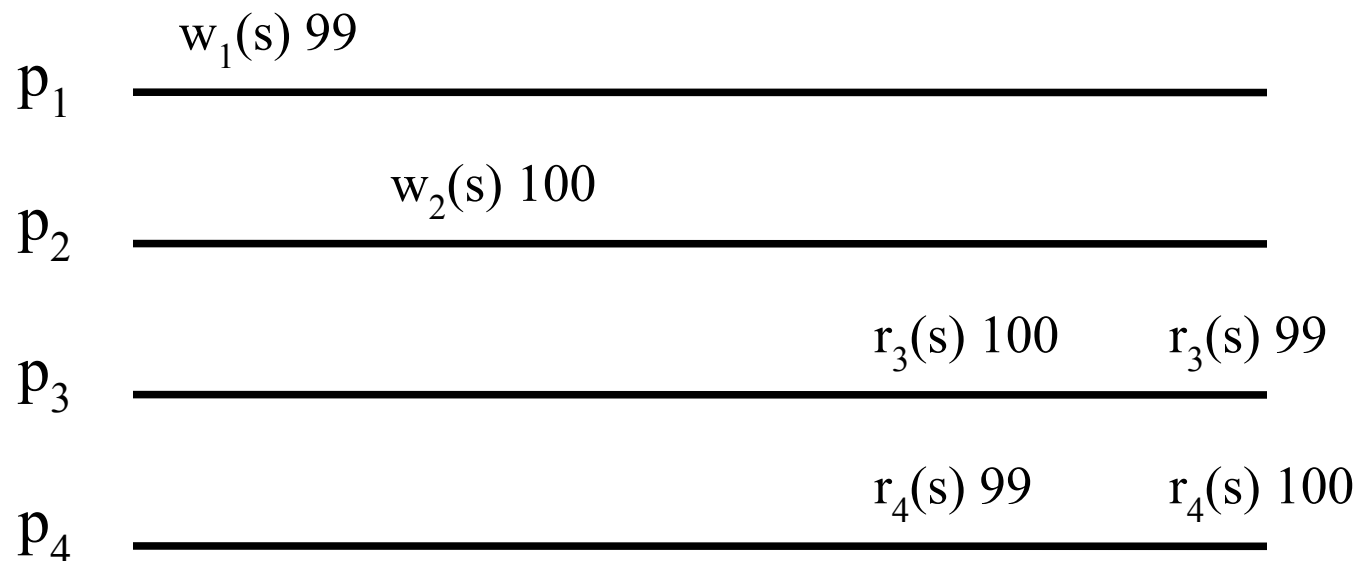


## ◆ Definition

- ◆ All writes that are (potentially) causally related must be seen by every process in the same order

## ◆ Example

- ◆ Is the following example causally consistent? **Yes!**
- ◆ Is the following example sequentially consistent? **No!**

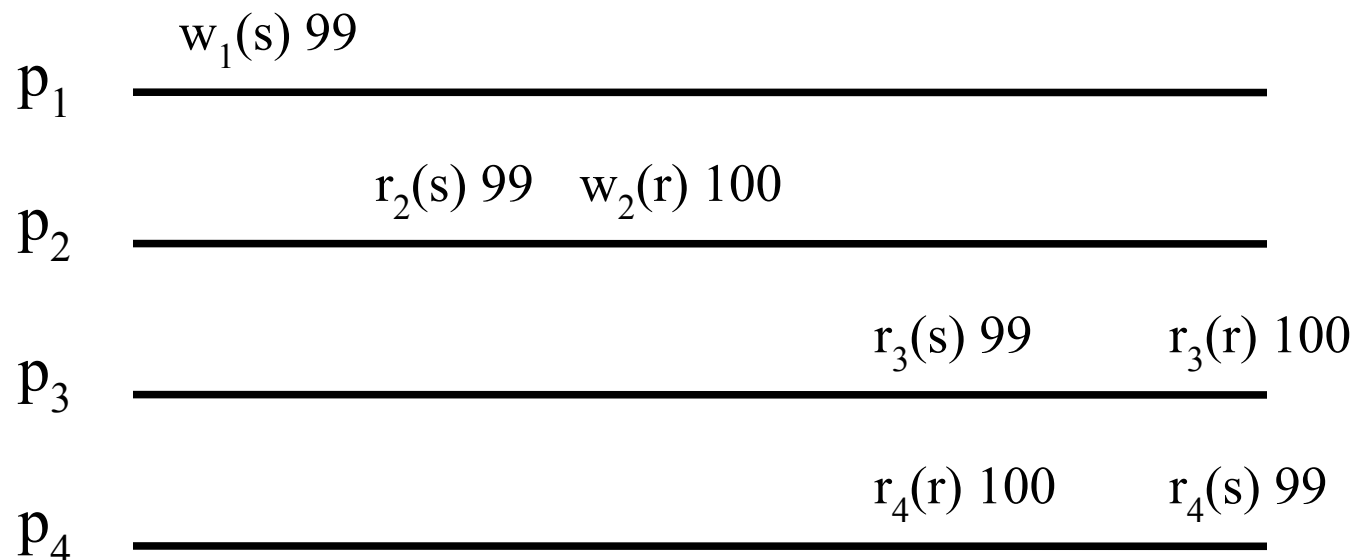


## ◆ Definition

- ◆ All writes that are (potentially) causally related must be seen by every process in the same order

## ◆ Example

- ◆ Is the following example causally consistent?
- ◆ Is the following example sequentially consistent?

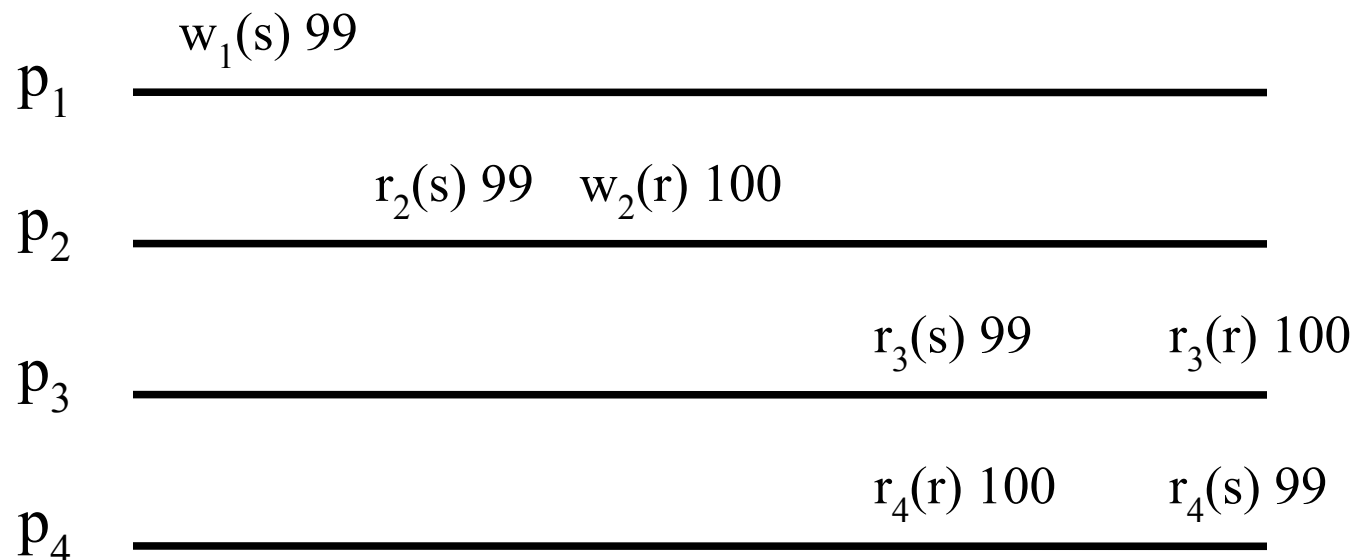


## ◆ Definition

- ◆ All writes that are (potentially) causally related must be seen by every process in the same order

## ◆ Example

- ◆ Is the following example causally consistent? **No!**
- ◆ Is the following example sequentially consistent?

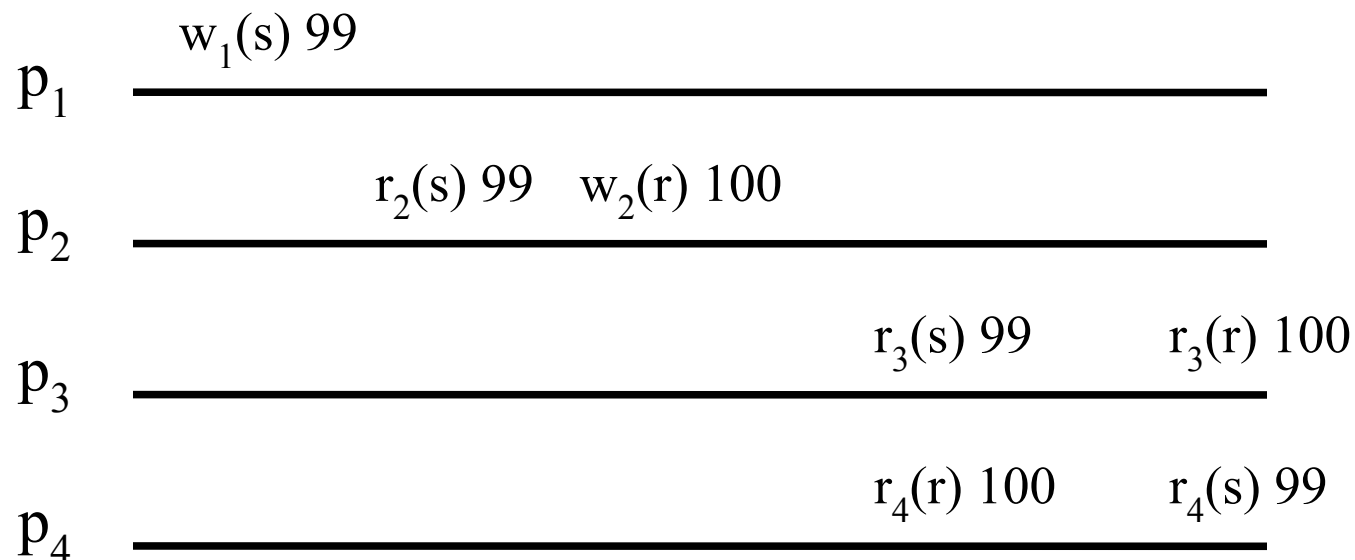


## ◆ Definition

- ◆ All writes that are (potentially) causally related must be seen by every process in the same order

## ◆ Example

- ◆ Is the following example causally consistent? **No!**
- ◆ Is the following example sequentially consistent? **No!**



## Additional consistency models

- ◆ **Other data-centric consistency models:**
  - ◆ FIFO/PRAM Consistency (Lipton and Sandberg, 1988)
  - ◆ Weak Consistency (Dubois et al, 1988)
  - ◆ Release Consistency (Gharachorloo et al, 1990)
  - ◆ Entry Consistency (Bershad et al, 1993)
- ◆ **We move now to client-centric consistency model**

# Adopting weaker forms of consistency

- ◆ **Scenario: consider a system where**
  - ◆ Updates are rare
  - ◆ Concurrent updates are absent, or can be easily resolved in an automatic way
  - ◆ Do we need sequential consistency in this case?
- ◆ **Example: DNS**
  - ◆ Updates are rare w.r.t. to reads!
  - ◆ Only a centralized authority can update the system; no concurrent updates.

# Adopting weaker forms of consistency

- ◆ **Definition: Eventual consistency**

- ◆ If no updates take place for a long time, all replicas will gradually become consistent (i.e., the same)

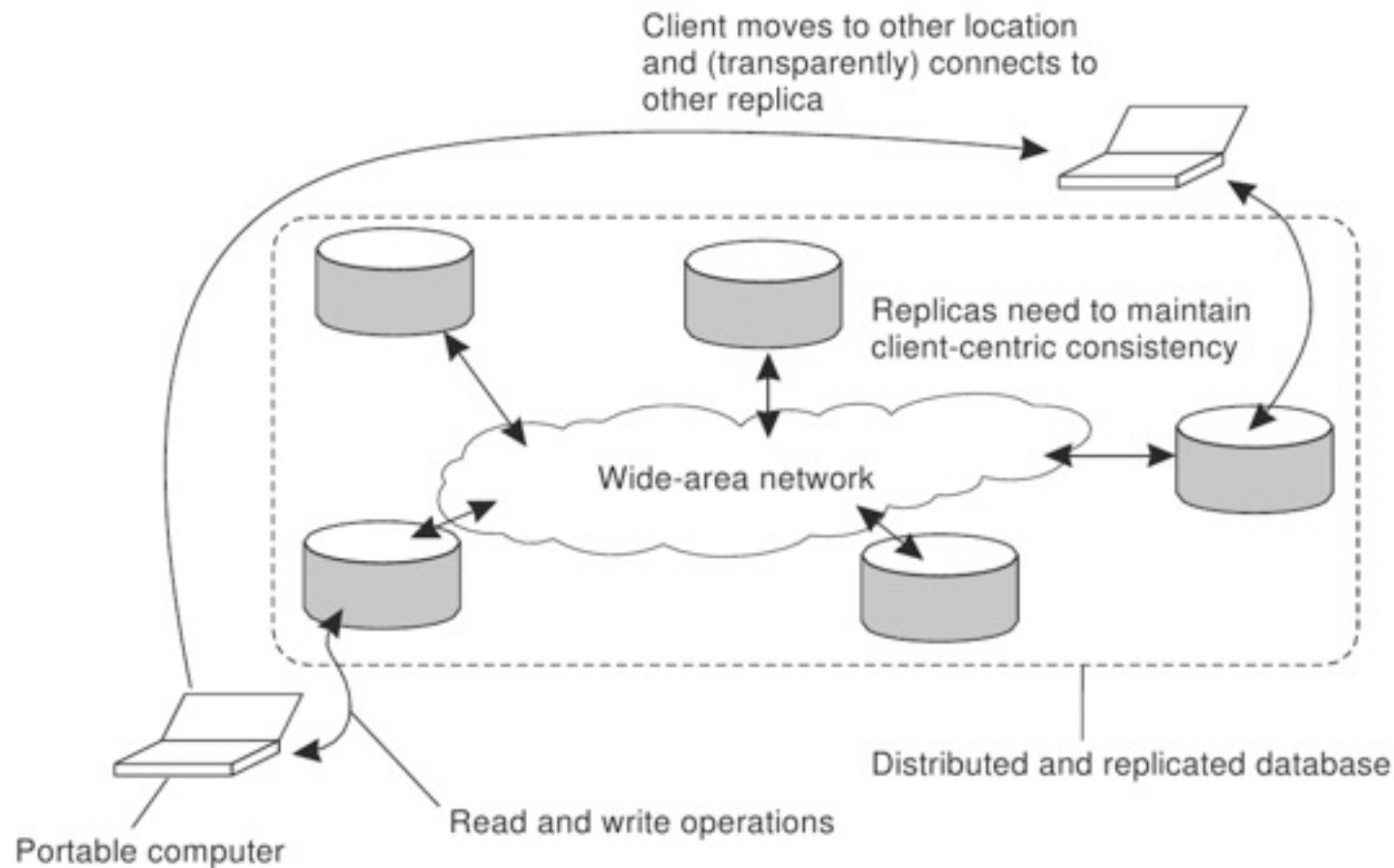
- ◆ **Questions**

- ◆ Can you suggest a protocol that provides eventual consistency?
- ◆ If you don't need sequential consistency, is eventual consistency always enough?

# Consistency for mobile users

## ◆ Example:

- ◆ Consider a replicated database that you access through your notebook. The notebook acts as a front-end to the database



# Consistency for mobile users

- ◆ **Problem: Eventual Consistency is not good**
  - ◆ You move from location  $A$  to location  $B$
  - ◆ Unless you use the same server, you may detect inconsistencies:
    - ◆ your updates at  $A$  may not have yet been propagated to  $B$
    - ◆ you may be reading newer entries than the ones available at  $A$
    - ◆ your updates at  $B$  may eventually conflict with those at  $A$
- ◆ **What we can do?**
  - ◆ The only thing you really care is that the entries you updated and/or read at  $A$ , are in  $B$  the way you left them in  $A$ . In that case, the database will appear to be consistent to *you*

# Client-centric consistency models

## ◆ Idea

- ◆ In some cases, we can avoid system-wide consistency, by concentrating on what specific clients want, instead of what should be maintained by servers

## ◆ Models

- ◆ Read-after-read / Monotonic reads
- ◆ Write-after-write / Monotonic writes
- ◆ Read-after-write / Read-your-writes
- ◆ Write-after-read / Write-follows-reads

## ◆ History:

- ◆ Bayou (replicated database, Terry et al., 1994)

# Client-centric consistency

## ◆ Notation

- ◆  $x_i[t]$  denotes the version of data item  $x$  at local copy  $L_i$  at time  $t$
- ◆  $x_i[t]$  is the result of a set of write operations on  $L_i$
- ◆ let  $WS(x_i[t])$  denote this set
- ◆ if operations in  $WS(x_i[t_1])$  have also been performed at local copy  $L_j$  at time  $t_2$ , we denote this with  $WS(x_i[t_1]; x_j[t_2])$
- ◆ **PS In the next figures, time is omitted**

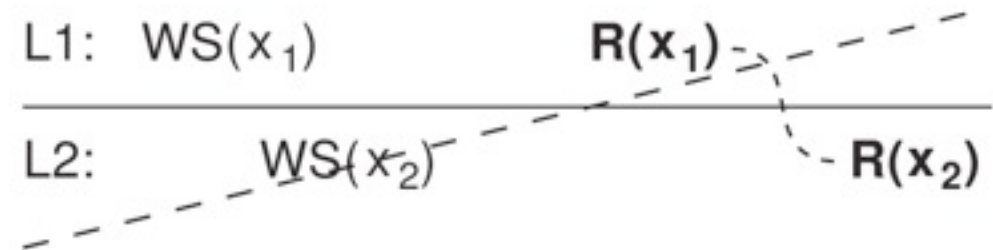
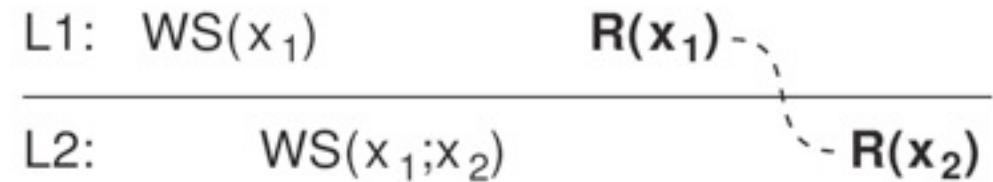
# Client-centric consistency

## ♦ Monotonic reads

- ♦ 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 or a more recent value

## ♦ Example:

- ♦ Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited



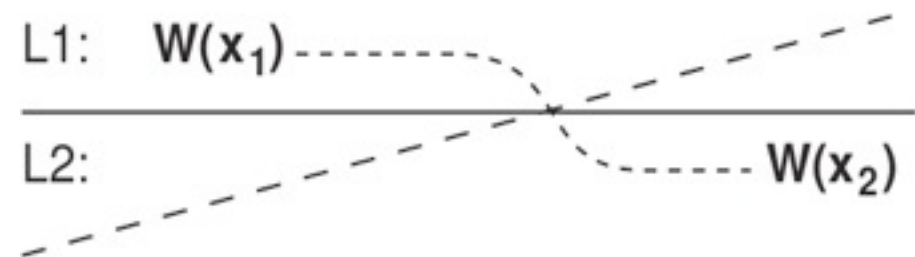
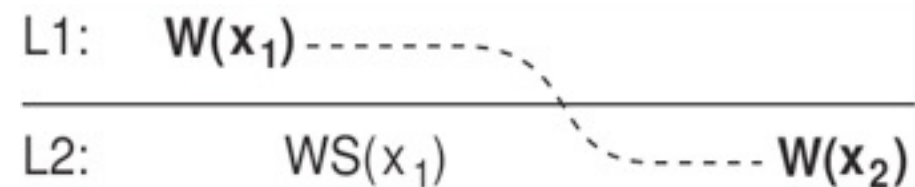
# Client-centric consistency

- ◆ **Monotonic writes:**

- ◆ A write operation by a process on a data item  $x$  is completed before any successive write operation on  $x$  by the same process

- ◆ **Example:**

- ◆ Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).



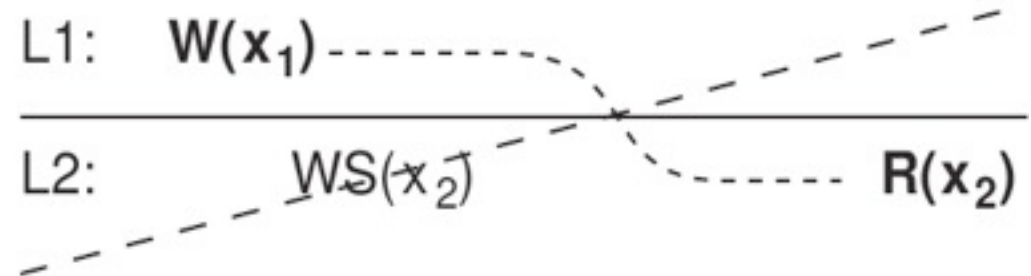
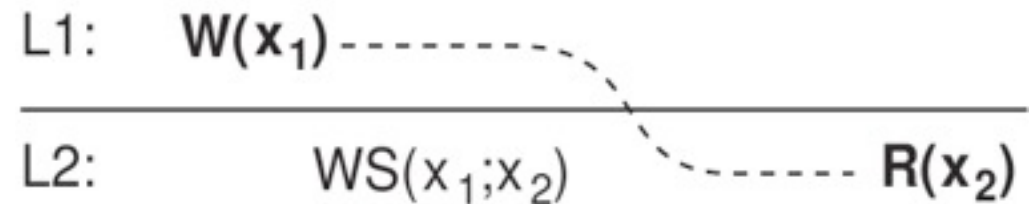
# Client-centric consistency

## ◆ Read-your-writes

- ◆ 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

## ◆ Example:

- ◆ Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.



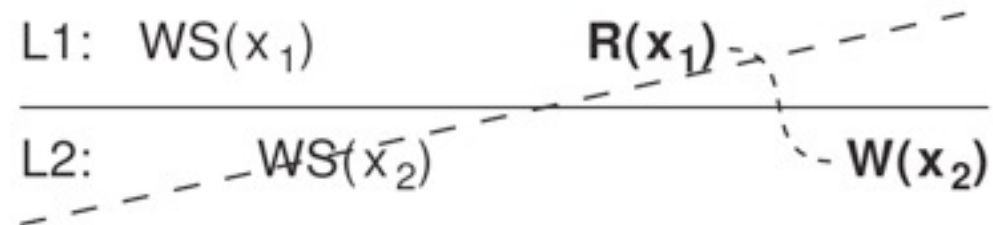
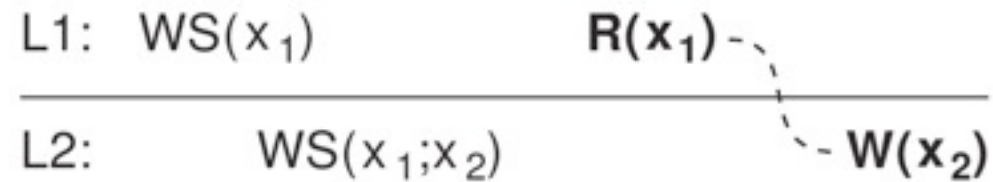
# Client-centric consistency

## ◆ Writes-follows-reads

- ◆ A write operation by a process  $P$  on a data item  $x$  following a previous read operation on  $x$  by  $P$ , is guaranteed to take place on the same or a more recent value of  $x$  that was read

## ◆ Example:

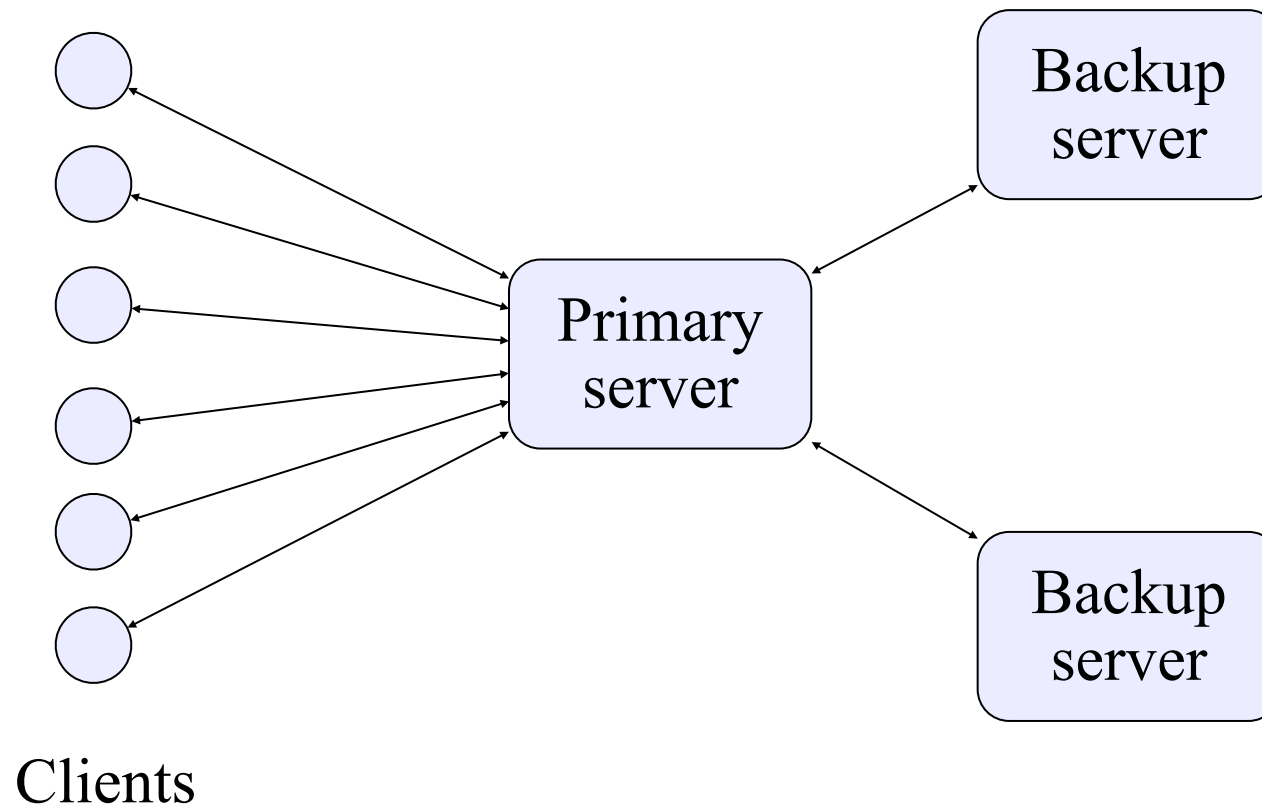
- ◆ Newsgroup:  
See reactions to posted articles only if you have the original posting  
(a read “pulls in” the corresponding write operation)



# Architecture of replicated data management

## ◆ Passive replication

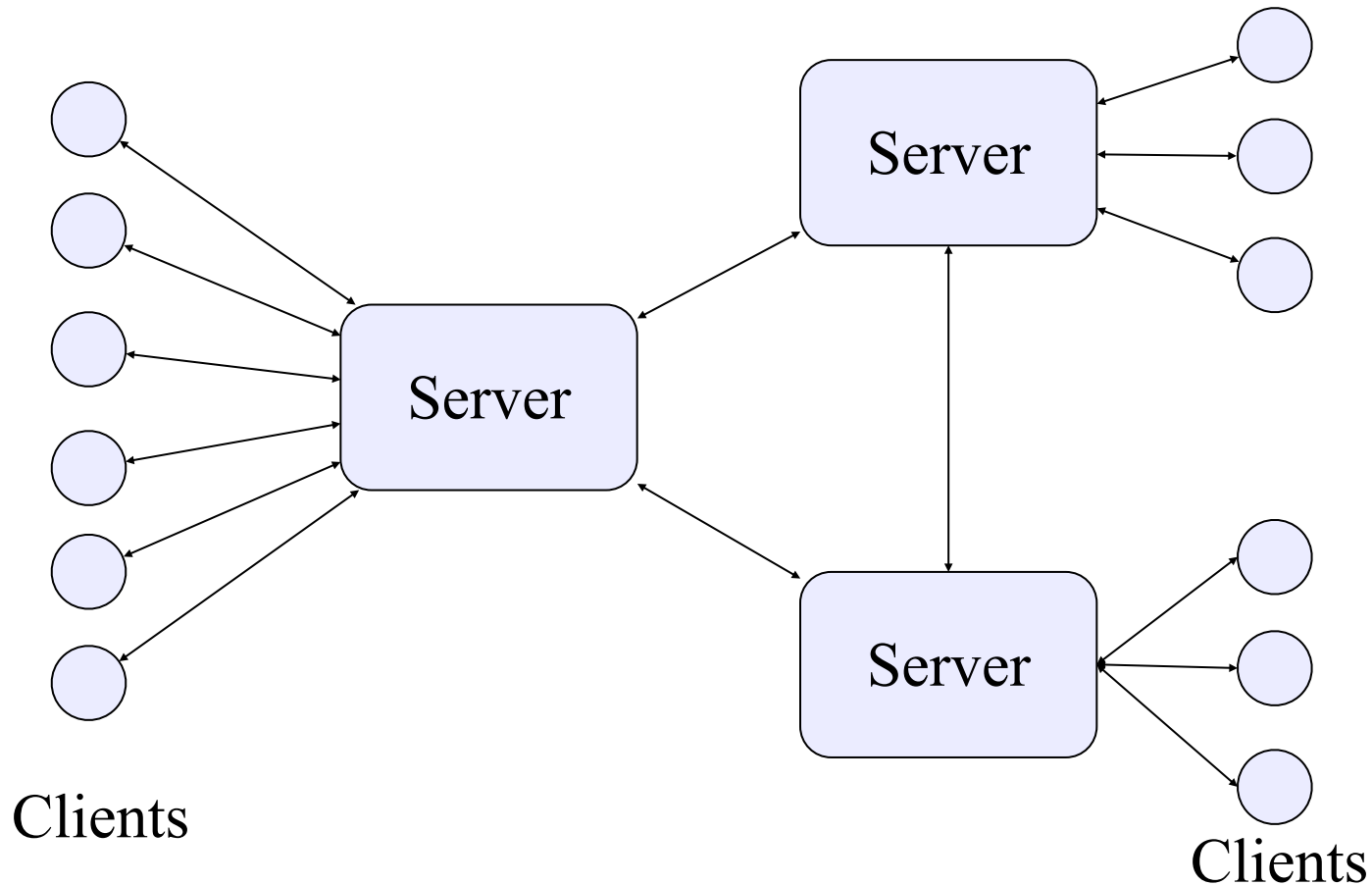
- ◆ Clients communicate with primary server
- ◆ Updates are forwarded from primary to backups
- ◆ Queries are replied by the primary



# Architecture of replicated data management

## ◆ Active replication

- ◆ All replicas handle the invocation and send the response
- ◆ Updates must be applied in the same order – total order broadcast



# Comparison between active and passive replication

## ◆ Active replication

- ◆ Uses more resources (CPU, bandwidth)
- ◆ More performance
- ◆ Deterministic (but see discussion later)

## ◆ Passive replication

- ◆ Computation is performed only at primary
- ◆ Primary becomes a bottleneck
- ◆ Can handle non-determinism

# Implementing linearizability – a general scheme

- ◆ **Assumptions**
  - ◆ Synchronous system
- ◆ **Initiator**
  - ◆ Passive: the primary
  - ◆ Active: the local replica
- ◆ **Implementation**
  - ◆ The initiator atomic-broadcasts all requests (read,write) to all servers
  - ◆ When the atomic-broadcast message is delivered at the initiator, it replies to the client
- ◆ **Correctness**
  - ◆ All replicas execute read,write in the same order

# Implementing sequential consistency – a general scheme

## ◆ Assumptions

- ◆ Synchronous system

## ◆ Initiator

- ◆ Passive: the primary
- ◆ Active: the local replica

## ◆ Implementation

- ◆ The initiator atomic-broadcasts write requests to all servers
- ◆ When the atomic-broadcast message is delivered, the replica updates its local copy
- ◆ Read request are replied immediately by the initiator

## ◆ Correctness

- ◆ Writes are executed in the same order everywhere
- ◆ Reads are consistent with local order

# Implementing causal consistency – a general scheme

## ◆ Assumptions

- ◆ Synchronous system

## ◆ Initiator

- ◆ Passive: the primary
- ◆ Active: the local replica

## ◆ Implementation

- ◆ The initiator causal-broadcasts write requests to all servers
- ◆ When the causal-broadcast message is delivered, the replica update its local copy
- ◆ Read request are replied immediately by the initiator

## ◆ Correctness

- ◆ Writes are executed in a causal order
- ◆ Reads are consistent with local (and causal) order

# Consistency protocols

- ◆ **Primary-based protocols**

- ◆ Definition
- ◆ Lower bounds

- ◆ **Replicated-write protocols**

- ◆ Majority, quorum-based
- ◆ State machine approach

- ◆ **Client-centric protocols**

- ◆ Monotonic reads, monotonic writes, read-your-writes, writes-follow-reads

# Primary-backup

- ◆ **The idea**
  - ◆ Clients communicate with a single replica (the **primary**)
  - ◆ The primary updates the other replicas (**backups**)
  - ◆ Backups detect the failure of the primary using a timeout mechanism
  - ◆ Clients learn from the service when the primary fails and the service “**fail over**” to a backup
- ◆ **Note: non-deterministic events are executed only at the primary**

# How to evaluate a primary-backup protocol

- ◆ **Degree of replication**

- ◆ number of servers used to implement the service; the smaller, the better

- ◆ **Blocking time**

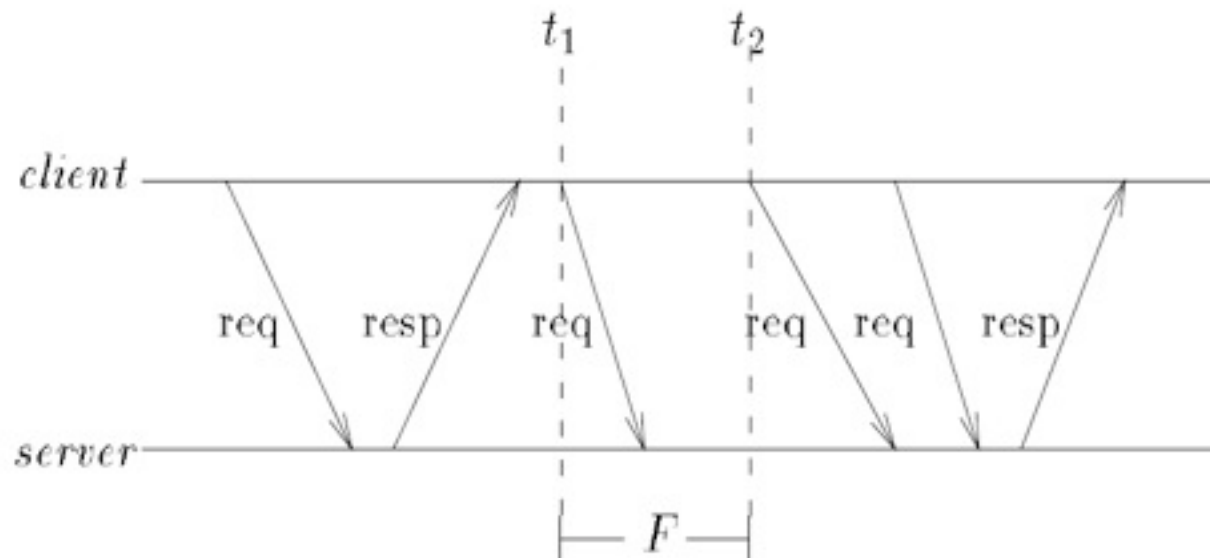
- ◆ the worst-case period between a request and its response in any failure-free execution

- ◆ **Failover time**

- ◆ the worst-case period during which request can be lost because there is no primary

# Definitions

- ◆ **Service outage:**
  - ◆ The service has a server outage at  $t$  if some correct client sends a request at time  $t$  to the service, but does not receive a response
- ◆  **$(k, \Delta)$ -bofo service - “bounded outage, finitely often”**
  - ◆ is one in which all server outages can be grouped into at most  $k$  intervals of time, each of at most length  $\Delta$



- ◆ **PB1:**
  - ◆ There exists a local predicate  $prmy_s$  on the state of each server  $s$ . At any time, there is at most one server  $s$  that satisfies  $prmy_s$
- ◆ **PB2:**
  - ◆ Each client  $i$  maintains a server identity  $Dest_i$  such that to make a request, client  $i$  sends a message to  $Dest_i$
- ◆ **PB3:**
  - ◆ If a client request arrives at a server that is not the current primary, then that request is not enqueued (and therefore is not processed)
- ◆ **PB4:**
  - ◆ There exist fixed values  $k$  and  $\Delta$  such that the service behaves like a single  $(k, \Delta)$ -bofo server

# Primary-backup – a simple protocol

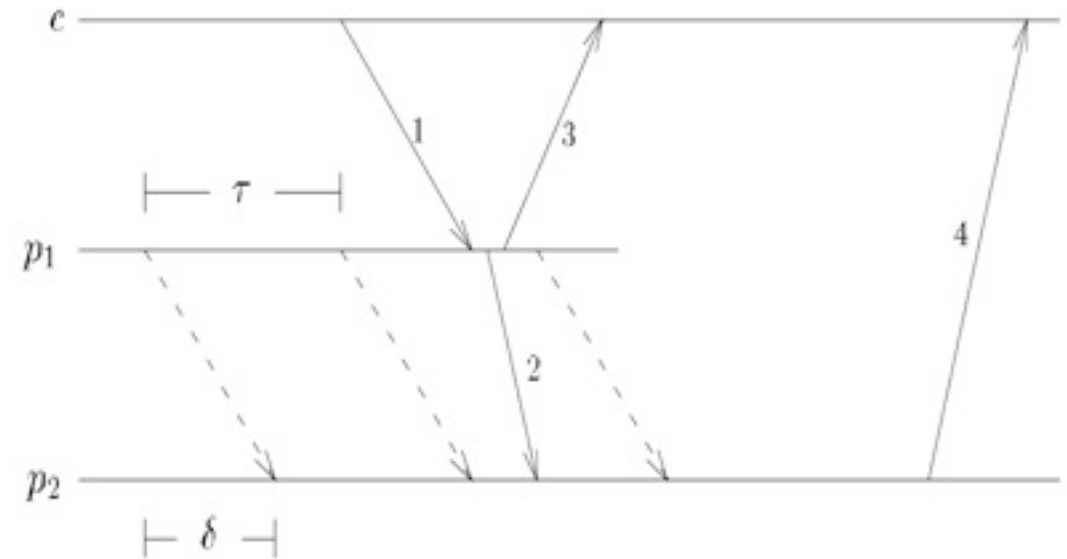
- ◆ We discuss now a simple protocol that implements the specification in the following system model
- ◆ **System model**
  - ◆ point-to-point communication
  - ◆ non-faulty channels
  - ◆ upper bound  $\delta$  on message delivery time
  - ◆ at most one server crashes
- ◆ **Two processes**
  - ◆ The primary  $p_1$
  - ◆ The backup  $p_2$

## Primary-backup – a simple protocol

- ◆ **On receipt of a client request, process  $p_1$** 
  - ◆ consumes request and updates its state
  - ◆ send state update message to  $p_2$
  - ◆ replies to client without waiting for ack from  $p_2$
- ◆ **Periodically, process  $p_1$** 
  - ◆ sends heartbeat message to  $p_2$  every  $\tau$  seconds
- ◆ **Upon receiving a state update from  $p_1$ , process  $p_2$** 
  - ◆ updates its state
- ◆ **If process  $p_2$  does not receive a heartbeat for  $\tau + \delta$  seconds,**
  - ◆  $p_2$  declares itself primary
  - ◆ it informs the clients
  - ◆ it begins consuming subsequent requests from clients

## ◆ PB1:

- ◆ There exists a local predicate  $prmy_s$  on the state of each server  $s$ . At any time, there is at most one server  $s$  that satisfies  $prmy_s$



## ◆ Proof

- ◆  $prmy_{p_1} = p_1$  has not crashed
- ◆  $prmy_{p_2} = p_2$  has not received a message from  $p_1$  for  $\tau + \delta$

## ◆ Failover time

- ◆  $\tau + 2\delta$

# Simple protocol – proof of correctness

## ◆ PB2:

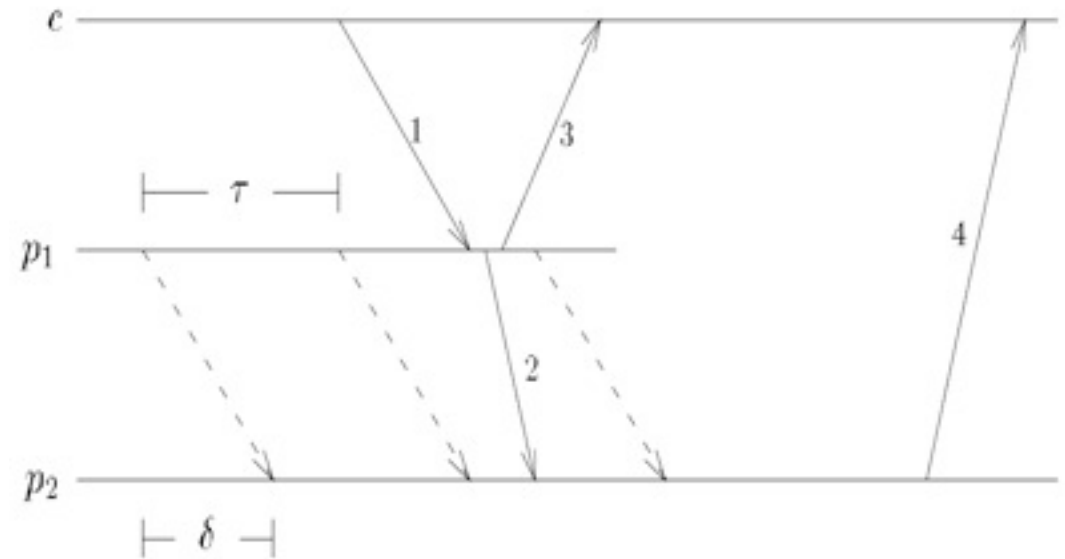
- ◆ Each client  $i$  maintains a server identity  $Dest_i$ , such that to make a request, client  $i$  sends a message to  $Dest_i$ ,

## ◆ PB3:

- ◆ If a client request arrives at a server that is not the current primary, then that request is not enqueued (and therefore is not processed)

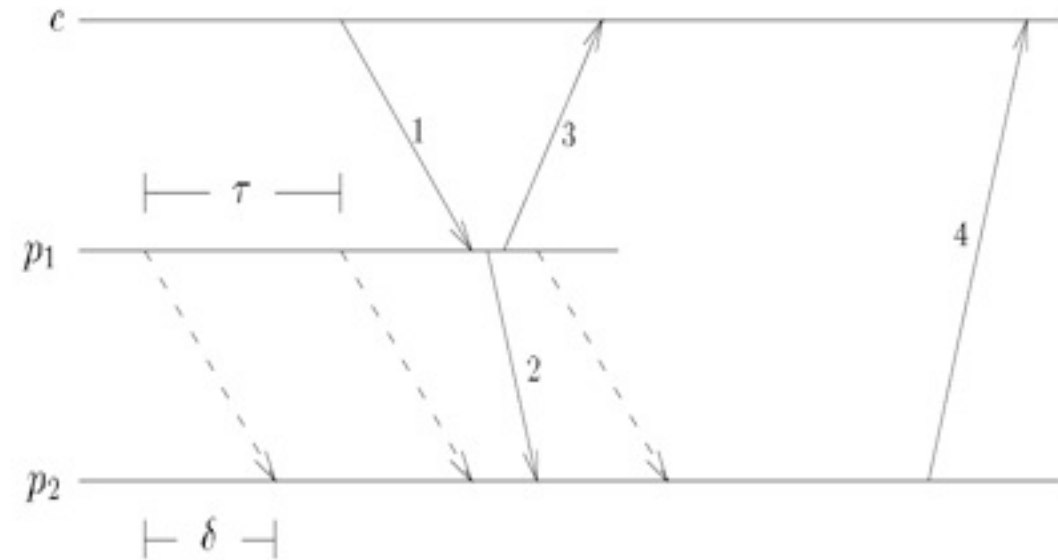
## ◆ Proof of correctness

- ◆ Trivially follows from the protocol



## Simple protocol – proof of correctness

- ◆ **PB4:**
  - ◆ There exist fixed values  $k, \Delta$  such that the service behaves like  $(k, \Delta)$ -bofo server
- ◆ **Proof of correctness: Find  $k, \Delta$**
- ◆ **At most one process can fail:  $k=1$**
- ◆  **$\Delta = \tau + 4\delta$** 
  - ◆ assume  $p_1$  crashes at  $t_c$
  - ◆ any client request sent to  $p_1$  at time  $t_c - \delta$  or later may be lost
  - ◆  $p_2$  may not become the new primary until  $t_c + \tau + 2\delta$
  - ◆ client may not learn that  $p_2$  is new primary for another  $\delta$



# Primary-backup – lower bounds

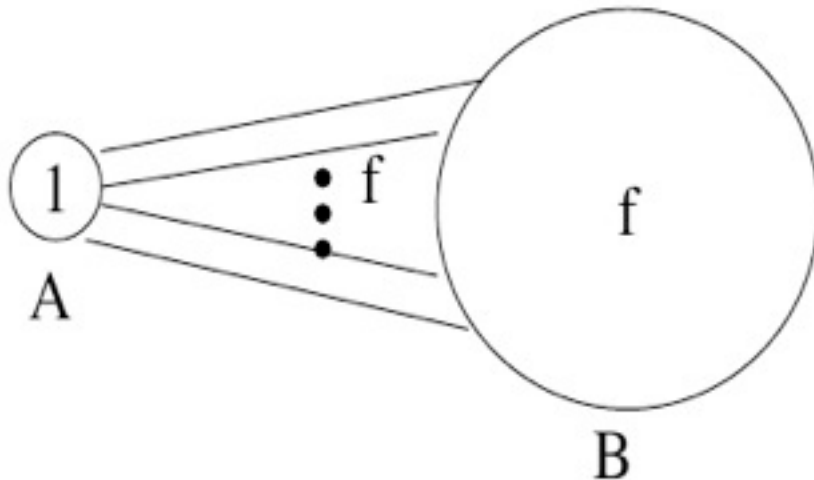
- ◆ **Fundamental question:**
  - ◆ Give that no more than  $f$  components can fail, what are the smallest possible values of the degree of replication, the blocking time and the failover time?
- ◆ **Lower bounds**
  - ◆ Knowing the lower bounds for a problem enables to evaluate the quality of a protocol
  - ◆ Tight lower bounds → optimal protocols
- ◆ **Example on components:**
  - ◆ Up to  $f$  crash+link failures → at most  $f$  processes may crash or  $f$  links may crash or  $f_1$  links +  $f_2$  processes =  $f$  components

# Lower bounds

## ◆ Degree of Replication

- ◆ crash  $n > f$
- ◆ crash + link  $n > f + 1$
- ◆ receive-omission  $n > \lfloor 3f/2 \rfloor$
- ◆ send-omission  $n > f$
- ◆ general-omission  $n > 2f$

## ◆ Example: crash+link



## ◆ Additional server by counter-example:

- ◆ suppose  $n = f + 1$
- ◆ divide the  $n$  servers in groups  $A$  and  $B$
- ◆ if all server in  $B$  crash,  $A$  must become primary
- ◆ if  $A$  crashes, one of the server of  $B$  must be primary
- ◆ what if all  $f$  links between  $A$  and  $B$  fails?

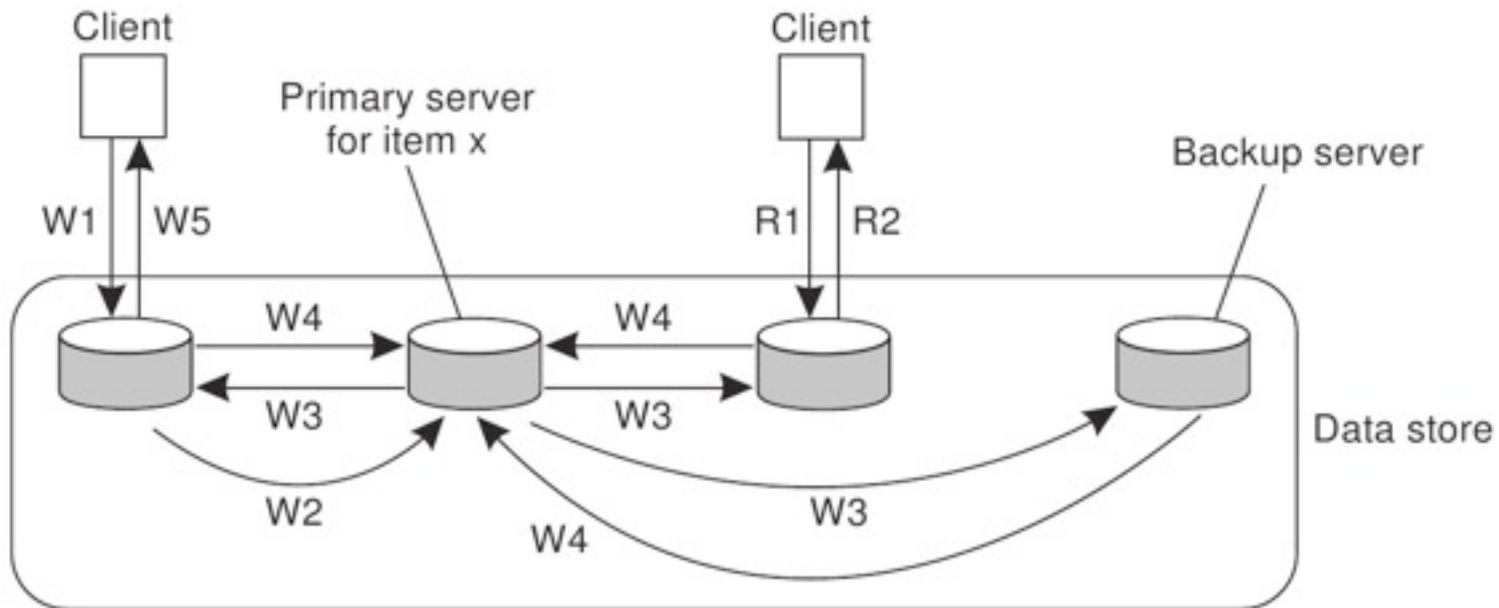
# Lower bounds

Failure Model	Degree of Replication	Blocking Time	Failover Time
crash	$n > f$	0	$f\delta$
crash+link	$n > f + 1$ †	0	$2f\delta$
receive-omission	$n > \lfloor \frac{3f}{2} \rfloor$ * †	$\delta$ when $n \leq 2f$ and $f = 1$ † $2\delta$ when $n \leq 2f$ and $f > 1$ * † $0$ when $n > 2f$	$2f\delta$
send-omission	$n > f$	$\delta$ when $f = 1$ $2\delta$ when $f > 1$	$2f\delta$
general-omission	$n > 2f$	$\delta$ when $f = 1$ $2\delta$ when $f > 1$	$2f\delta$

\* Bound not known to be tight.

# Primary-backup – more complex examples

## Multiple primaries

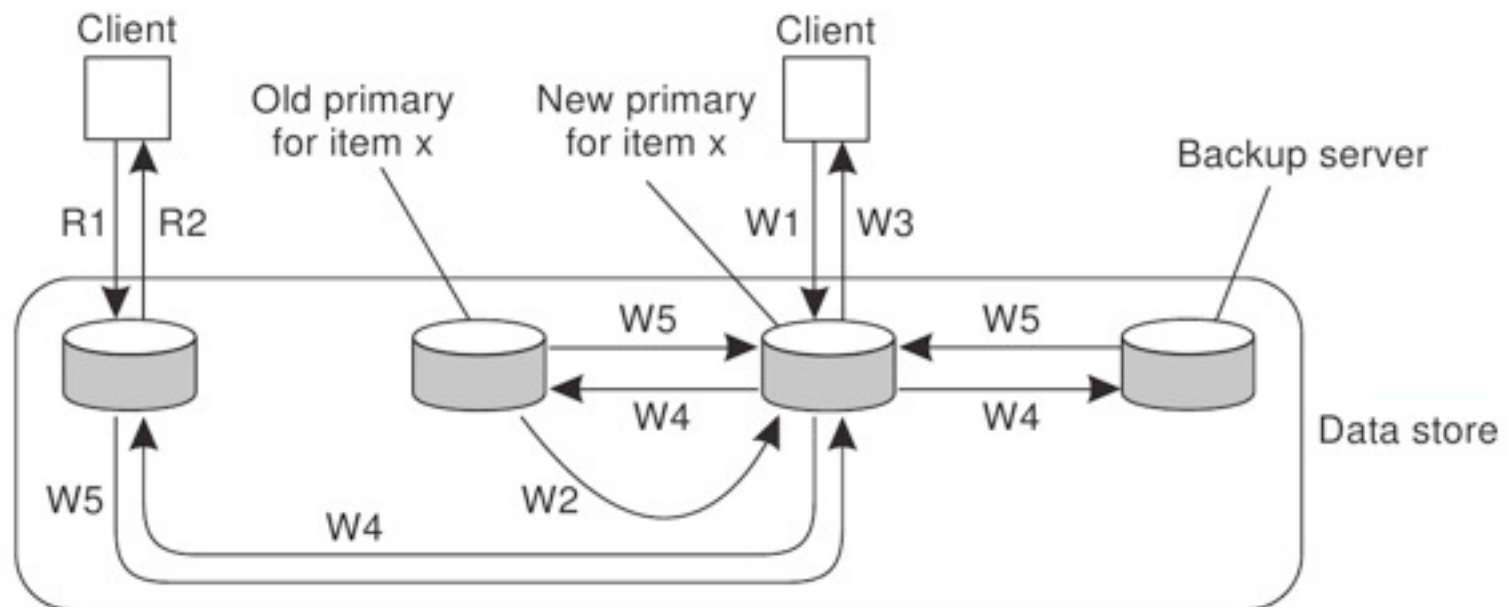


W1. Write request  
W2. Forward request to primary  
W3. Tell backups to update  
W4. Acknowledge update  
W5. Acknowledge write completed

R1. Read request  
R2. Response to read

## Primary-backup – more complex examples

**Local writes: Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).**



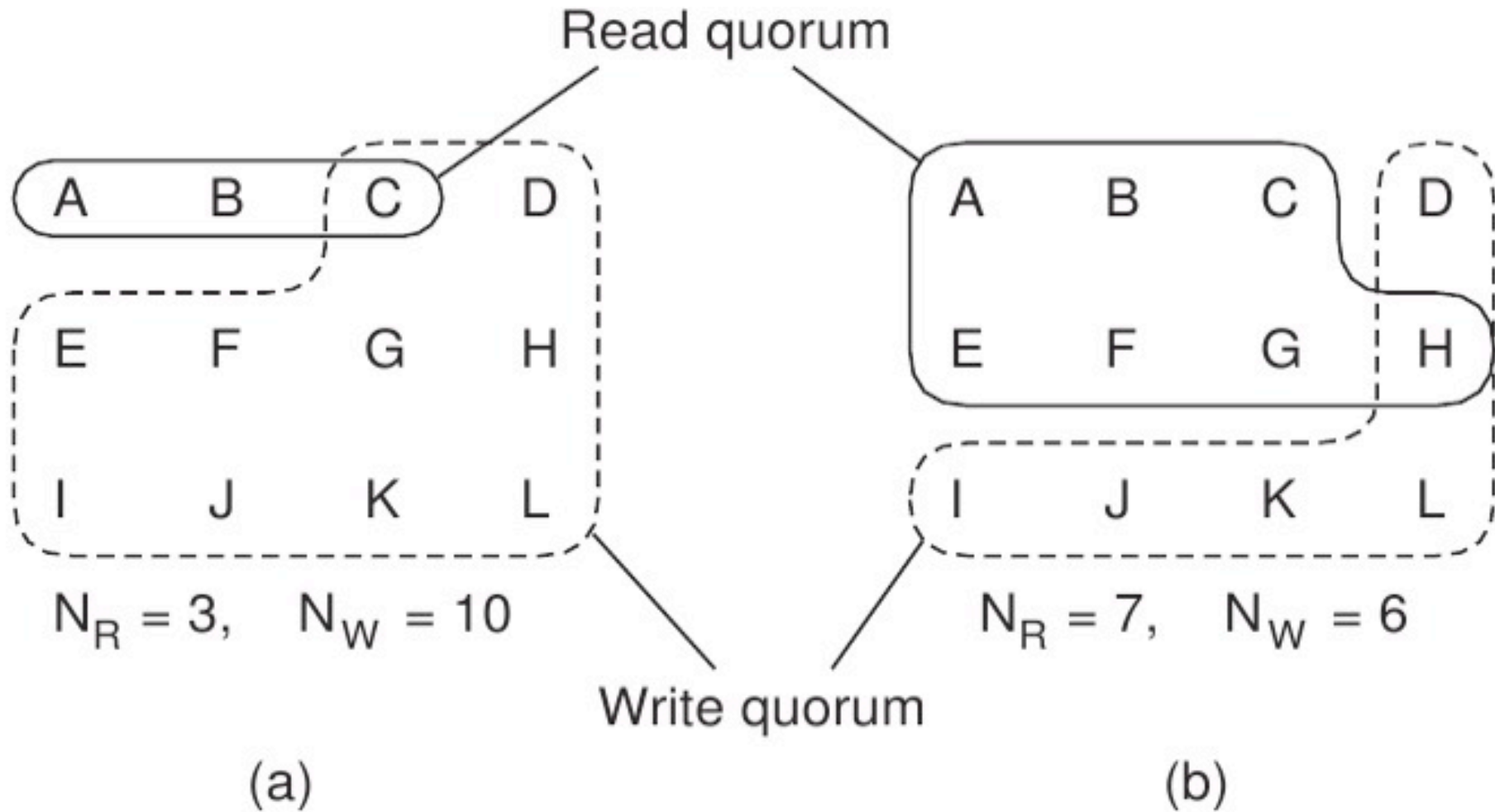
W1. Write request  
W2. Move item x to new primary  
W3. Acknowledge write completed  
W4. Tell backups to update  
W5. Acknowledge update

R1. Read request  
R2. Response to read

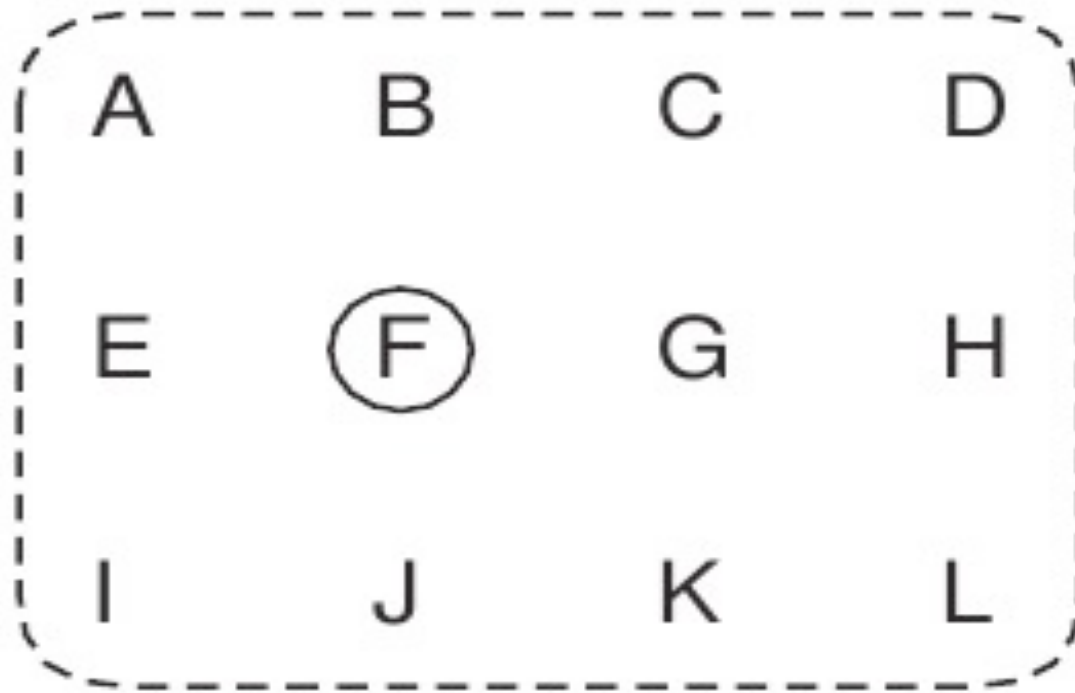
## Replicated-write protocols: Quorum Based

- ◆ **Quorum-based protocols: Ensure that each operation is carried out in such a way that a majority vote is established**
  - ◆ Read quorum  $n_R$
  - ◆ Write quorum  $n_W$
  - ◆ Constraints
    - ◆  $n_R + n_W > n$  (prevent read-write conflicts)
    - ◆  $n_W > n/2$  (prevent write-write conflicts)
  - ◆ When you do a read, you take the most up-to-date entry
  - ◆ Quorums guarantee that the last written entry will be present

# Quorum examples



# Quorum example



$$N_R = 1, \quad N_W = 12$$

# Client-centric consistency

## ◆ Naive implementation

- ◆ Each write operation is assigned a unique identifier
  - ◆ Done by the server where the operation is requested
- ◆ For each client  $c$ , we keep track of:
  - ◆ Read set  $WS_R$ : contains write operations relevant to the read operations performed by  $c$
  - ◆ Write set  $WS_W$ : contains write operations relevant to the write operations performed by  $c$
- ◆ For each server, we keep track
  - ◆ Write set  $WS$ : contains the write operations executed so far

# Client-centric consistency

- ◆ **Monotonic-read, naive implementation**
  - ◆ To perform a read operation  $r_c$ , a client  $c$  must
    - ◆ send its read set  $WS_R$  to the server
  - ◆ The server
    - ◆ Check whether all the writes in  $WS_R$  have been executed locally
    - ◆ If not, ask the appropriate servers the missing operations
    - ◆ Return the requested value to the client and its  $WS$  set
  - ◆ The client
    - ◆ Add  $WS$  to its local read set:  $WS_R = WS_R \cup WS$

# Client-centric consistency

- ◆ **Monotonic-write, naive implementation**
  - ◆ To perform a **write** operation  $w_c$ , a client  $c$  must
    - ◆ send its read set  $WS_w$  to the server
  - ◆ The server
    - ◆ Check whether all the writes in  $WS_w$  have been executed locally
    - ◆ If not, ask the appropriate servers the missing operations
    - ◆ **Applies the write**
    - ◆ **Add the operation to  $WS$**
    - ◆ **Send  $WS$  back to the client**
  - ◆ The client
    - ◆ Add  $WS$  to its local write set:  $WS_w = WS_w \cup WS$

# Client-centric consistency

- ◆ **Improving efficiency**
  - ◆ Operations are grouped in sessions
  - ◆ Once a session ends, the read/write sets are reseted
  - ◆ We use vector clocks to represent sets
    - ◆ Each entry: last write operation known to be executed by a server

# State machine

- ◆ **Set of state variables + Sequence of commands**
- ◆ **A command**
  - ◆ Reads its read set values (rsv)
  - ◆ Writes to its write set values (wsv)
- ◆ **A deterministic command**
  - ◆ Produces deterministic wsvs and outputs on given rsv
- ◆ **A deterministic state machine**
  - ◆ Reads a fixed sequence of deterministic commands

# State machine

- ◆ **Replica coordination: All non-faulty state machines receive all commands in the same order**
- ◆ **Agreement:**
  - ◆ Every non-faulty state machine receives every command
- ◆ **Order:**
  - ◆ Every non-faulty state machine processes the commands it receives in the same order
- ◆ **Note:**
  - ◆ Only write request need to be executed by all (sequential consistency)
  - ◆ Commutative operations do not need to be ordered

# Where should RC be implemented?

- ◆ **In hardware**
  - ◆ sensitive to architecture changes
- ◆ **At the OS level**
  - ◆ state transitions hard to track and coordinate
- ◆ **At the application level**
  - ◆ requires sophisticated application programmers
- ◆ **The answer:**
  - ◆ Virtual machines

# Hypervisor-based fault tolerance

- ◆ **Implement RC at a virtual machine running on the same instruction-set as underlying hardware**
- ◆ **Undetectable by higher layers of software**
- ◆ **One of the great come-backs in systems research!**
  - ◆ CP-67 for IBM 369 [1970]
  - ◆ Xen [SOSP 2003], VMware
- ◆ **Two types of commands**
  - ◆ virtual-machine instructions
  - ◆ virtual-machine interrupts (with DMA input)
- ◆ **State transition must be deterministic**
  - ◆ ...but some VM instructions are not (e.g. time-of-day)
  - ◆ interrupts must be delivered at the same point in cmd sequence

## Hypervisor approach - history

- ◆ **Thomas C. Bressoud, Fred B. Schneider. *Hypervisor-based Fault Tolerance*. ACM TOCS, 14(1):80-107**
  - ◆ The work (partially) presented here
- ◆ **John R. Douceur and Jon Howell. *Replicated Virtual Machines*. Microsoft Research TR-2005-119**
  - ◆ Technical paper associated to a patent
- ◆ **Brendan Cully et al. *Remus: High Availability via Asynchronous Virtual Machine Replication*. NSDI'08.**
  - ◆ Best paper award
  - ◆ Real implementation for XEN

# Hypervisor-based fault-tolerance

- ◆ **Ordinary (deterministic) instructions**
  - ◆ Example: processor instructions
  - ◆ How implemented? Executed directly by the processor
- ◆ **Environment (nondeterministic) instructions**
  - ◆ Example: get-time-of-day()
  - ◆ **Environment Instruction Assumption**
    - ◆ Hypervisor captures all environment instructions, simulates them, and ensures they have the same effect at all state machines
  - ◆ How implemented? By capturing system calls

- ◆ **VM interrupts must be delivered at same point in instruction sequence at all replicas**
  - ◆ Example: reply from DMA devices
  - ◆ **Instruction Stream Interrupt Assumption**
  - ◆ Hypervisor can be invoked at specific point in the instruction stream
  - ◆ How implemented?
    - ◆ **Recovery register**
      - ◆ counts the number of instructions, then interrupt
      - ◆ separates instruction stream into epochs
      - ◆ In each epoch, primary VM buffer interrupts, and forward these to backup VM in the epoch's end
      - ◆ Interrupts at backup VM are ignored

## Algorithm (primary, failure-free)

- ◆ **P0 : if primary VM execute Env. Instruction at pc**
  - ◆ Send  $[E(p), pc, Val]$  to backup and wait for ack
- ◆ **P1 : if primary VM receives a interrupt**
  - ◆ Buffer interrupt for delivery later.
- ◆ **P2 : if primary's epoch ends**
  - ◆ Primary send to backup all buffered interrupts during  $E(p)$  and wait for ack.
  - ◆ Primary delivers all interrupts
  - ◆  $E(p) = E(p) + 1$  , and primary start epoch  $E(p)$  .

## Algorithm (backup, failure free)

- ◆ **P3 : if backup VM execute Env. Instruction at pc**
  - ◆ Wait receipt of  $[E(b), pc, Val]$  from primary
  - ◆ If  $[E(b), pc, Val]$  is received, then ack primary
- ◆ **P4 : if backup VM receives a interrupt**
  - ◆ It's ignored.
- ◆ **P5 : if epoch ends**
  - ◆ Wait for all buffered interrupts from primary, if received, then ack primary.
  - ◆ Backup VM delivers all interrupts.
  - ◆  $E(b) = E(b) + 1$  , and backup start epoch  $E(b)$  .

## Algorithm (failures)

- ◆ **If b receives a failure notification in P3 instead of  $[e(b), pc, Val_i]$** 
  - ◆ b executes i
- ◆ **If b receives failure notification in P5 instead of interrupts:**
  - ◆  $e(b) := e(b) + 1$
  - ◆ b starts  $e(b)$  <--- failover epoch
  - ◆ b is promoted primary for epoch  $e(b) + 1$