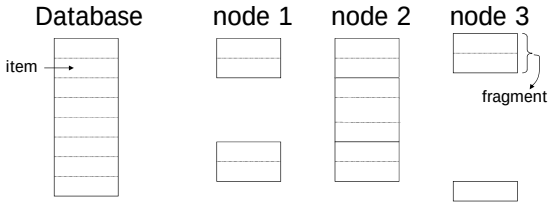
CS 347:
Parallel and Distributed
Data Management
Notes07: Data Replication

Hector Garcia-Molina

Data Replication

- Reliable net, fail-stop nodes
- The model



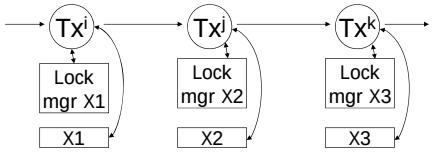
- Study one fragment, for time being
- Data replication ⇒ higher availability

Outline

- Basic Algorithms
- Improved (Higher Availability) Algorithms
- Multiple Fragments & Other Issues

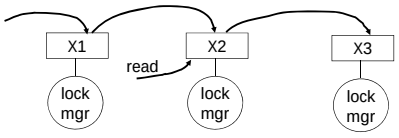
Basic Solution (for C.C.)

- Treat each copy as an independent data item

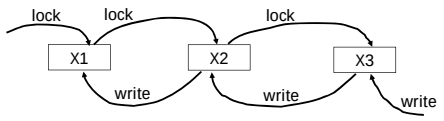


Object X has copies X1, X2, X3

- Read(X):
  - get shared X1 lock
  - get shared X2 lock
  - get shared X3 lock
  - read one of X1, X2, X3
  - at end of transaction, release X1, X2, X3 locks



- Write(X):
  - get exclusive X1 lock
  - get exclusive X2 lock
  - get exclusive X3 lock
  - write new value into X1, X2, X3
  - at end of transaction, release X1, X2, X3 locks

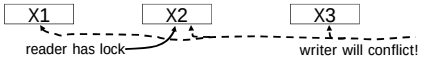


- Correctness OK
  - 2PL ⇒ serializability
  - 2PC ⇒ atomic transactions
- Problem: Low availability

☒ cannot access X!

Basic Solution — Improvement

- Readers lock and access a single copy
- Writers lock all copies
  and update all copies

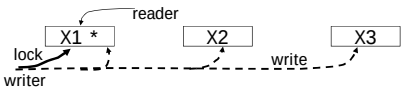


- Good availability for reads
- Poor availability for writes

Reminder

- With basic solution
  - use standard 2PL
  - use standard commit protocols

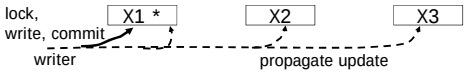
Variation on Basic: Primary copy



- Select primary site (static for now)
- Readers lock and access primary copy
- Writers lock primary copy
  and update all copies

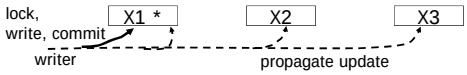
Commit Options for Primary Site Scheme

- Local Commit



Commit Options for Primary Site Scheme

- Local Commit



Write(X):
- Get exclusive X1* lock
- Write new value into X1*

⋮

- Commit at primary; get sequence number

⋮

- Perform X2, X3 updates in sequence number order

Example t = 0

X1: 0 *
Y1: 0
Z1: 0

X2: 0
Y2: 0
Z2: 0

T1: X ← 1; Y ← 1;
T2: Y ← 2;
T3: Z ← 3;

Example t = 1

X1: 1 *
Y1: 1
Z1: 3

X2: 0
Y2: 0
Z2: 0

T1: X ← 1; Y ← 1; ← active at node 1
T2: Y ← 2; ← waiting for lock at node 1
T3: Z ← 3; ← active at node 1

Example t = 2

X1: 1 *
Y1: 2
Z1: 3

X2: 0
Y2: 0
Z2: 0

#2: X ← 1; Y ← 1

T1: X ← 1; Y ← 1; ← committed
T2: Y ← 2; ← active at 1
T3: Z ← 3; ← committed

#1: Z ← 3

Example t = 3

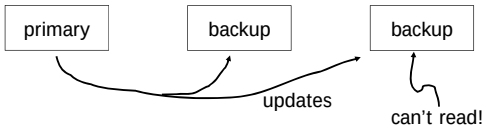
X1: 1 *
Y1: 2
Z1: 3

X2: 1
Y2: 2
Z2: 3

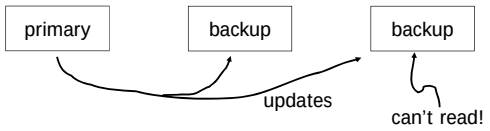
#1: Z ← 3
#2: X ← 1; Y ← 1
#3: Y ← 2

T1: X ← 1; Y ← 1; ← committed
T2: Y ← 2; ← committed
T3: Z ← 3; ← committed

What good is RPWP-LC?



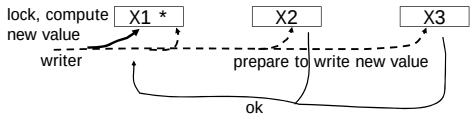
What good is RPWP-LC?



Answer: Can read "out-of-date" backup copy
(also useful with 1-safe backups... later)

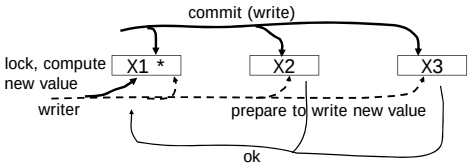
Commit Options for Primary Site Scheme

- Distributed Commit



Commit Options for Primary Site Scheme

- Distributed Commit



Example

X1: 0 *
Y1: 0
Z1: 0

X2: 0
Y2: 0
Z2: 0

T1: X ← 1; Y ← 1;
T2: Y ← 2;
T3: Z ← 3;

Basic Solution

- Read lock all; write lock all: RAWA
- Read lock one; write lock all: ROWA
- Read and write lock primary: RPWP
  - local commit: LC
  - distributed commit: DC

Comparison

N = number of nodes with copies
P = probability that a node is operational

	Probability can read	Probability can write
RAWA		
ROWA		
RPWP:LC		
RPWP:DC		

Comparison

N = number of nodes with copies
P = probability that a node is operational

	Probability can read	Probability can write
RAWA	P^N	P^N
ROWA	$1 - (1-P)^N$	P^N
RPWP:LC	P	P
RPWP:DC	P	P^N

Comparison

N = 5 = number of nodes with copies
P = 0.99 = probability that a node is operational

	Read Prob.	Write Prob.
RAWA	0.9510	0.9510
ROWA	1.0000	0.9510
RPWP:LC	0.9900	0.9900
RPWP:DC	0.9900	0.9510

Comparison

N = 100 = number of nodes with copies
P = 0.99 = probability that a node is operational

	Read Prob.	Write Prob.
RAWA	0.3660	0.3660
ROWA	1.0000	0.3660
RPWP:LC	0.9900	0.9900
RPWP:DC	0.9900	0.3660

Comparison

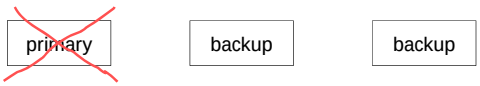
N = 5 = number of nodes with copies
P = 0.90 = probability that a node is operational

	Read Prob.	Write Prob.
RAWA	0.5905	0.5905
ROWA	1.0000	0.5905
RPWP:LC	0.9000	0.9000
RPWP:DC	0.9000	0.5905

Outline

- Basic Algorithms
- Improved (Higher Availability) Algorithms
  - Mobile Primary
  - Available Copies
- Multiple Fragments & Other Issues

Mobile Primary (with RPWP)



(1) Elect new primary
(2) Ensure new primary has seen all previously committed transactions
(3) Resolve pending transactions
(4) Resume processing

(1) Elections

- Can be tricky...
- One idea:
  - Nodes have IDs
  - Largest ID wins

(1) Elections: One scheme

(a) Broadcast "I want to be primary, ID=X"

(b) Wait long enough so anyone with larger ID can stop my takeover

(c) If I see "I want to be primary" message with smaller ID, kill that takeover

(d) After wait without seeing bigger ID, I am new primary!

(1) Elections: Epoch Number

It is useful to attach an epoch or version number to messages:

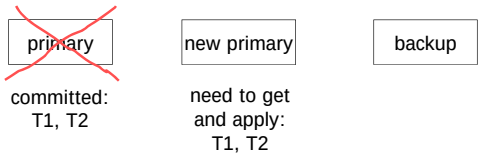
primary: n3 epoch# = 1

primary: n5 epoch# = 2

primary: n3 epoch# = 3

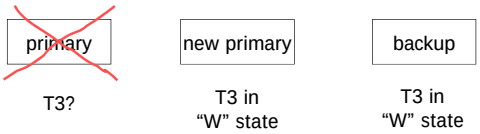
...

(2) Ensure new primary has previously committed transactions

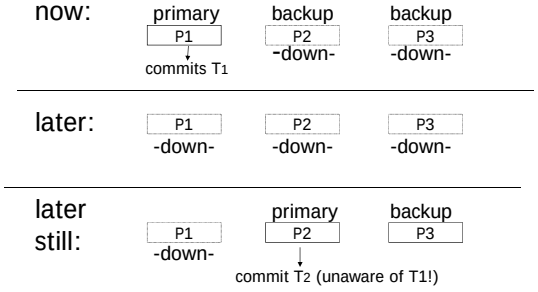


⇒ How can we make sure new primary is up to date? More on this coming up...

(3) Resolve pending transactions



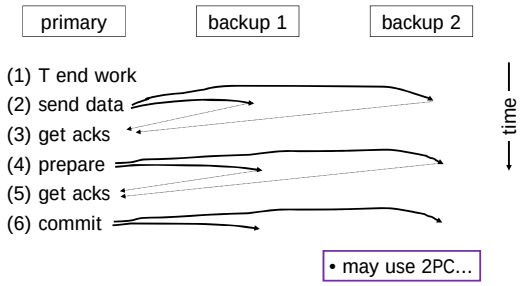
Failed Nodes: Example



RPWP:DC & 3PC take care of problem!

- Option A
  - Failed node waits for:
    - commit info from active node, or
    - all nodes are up and recovering
- Option B
  - Majority voting

RPWP: DC & 3PC



- may use 2PC…

Node Recovery

- All transactions have commit sequence number
- Active nodes save update values "as long as necessary"
- Recovering node asks active primary for missed updates; applies in order

Example: Majority Commit

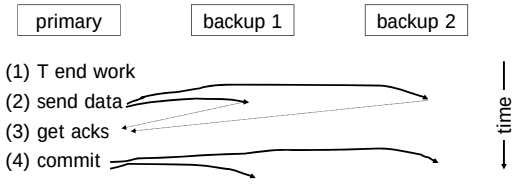
state:	C1	C2	C3
C	T1,T2,T3	T1,T2	T1,T3
P		T3	T2
W	T4	T4	T4

Example: Majority Commit

t1: C1 fails
t2: C2 new primary
t3: C2 commits T1, T2, T3; aborts T4
t4: C2 resumes processing
t5: C2 commits T5, T6
t6: C1 recovers; asks C2 for latest state
t7: C2 sends committed and pending transactions; C2 involves C1 in any future transactions

2-safe vs. 1-safe Backups

- Up to now we have covered 3/2-safe backups (RPWP:DC):



Guarantee

- After transaction T commits at primary, any future primary will "see" T

now:

primary | backup 1 | backup 2

T1, T2, T3

later:

~~primary~~ | next primary | backup 2

T1, T2, T3, T4

Performance Hit

- 3PC is very expensive
  - many messages
  - locks held longer (less concurrency) [Note: group commit may help]
- Can use 2PC
  - may have blocking
  - 2PC still expensive [up to 1 second reported]

Alternative: 1-safe (RPWP:LC)

- Commit transactions unilaterally at primary
- Send updates to backups as soon as possible

primary | backup 1 | backup 2

(1) T end work
(2) T commit
(3) send data
(4) purge data

time

Problem: Lost Transactions

now:

primary	backup 1	backup 2
T1, T2, T3	T1	T1

later:

~~primary~~	next primary	backup 2
	T1, T4, T5	T1, T4

Claim

- Lost transaction problem tolerable
  - failures rare
  - only a "few" transactions lost

Primary Recovery with 1-safe

- When failed primary recovers, need to "compensate" for missed transactions

now:

~~primary~~	next primary	backup 2
T1, T2, T3	T1, T4, T5	T1, T4

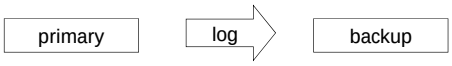
later:

backup 3	next primary	backup 2
T1, T2, T3, $T3^{-1}$, $T2^{-1}$, T4, T5	T1, T4, T5	T1, T4, T5

compensation

Log Shipping

- "Log shipping:" propagate updates to backup



- Backup replays log
- How to replay log efficiently?
  - e.g., elevator disk sweeps
  - e.g., avoid overwrites

So Far in Data Replication

- RAWA
- ROWA
- Primary copy
  - static
    - local commit
    - distributed commit
  - mobile primary
    - 2-safe (distributed commit) blocking or non-blocking
    - 1-safe (local commit)

Outline

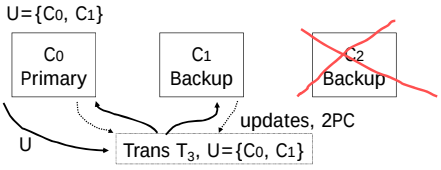
- Basic Algorithms
- Improved (Higher Availability) Algorithms
  - Mobile Primary
  - Available Copies
- Multiple Fragments & Other Issues

PC-lock available copies

- Transactions write lock at all available copies
- Transactions read lock at any available copy
- Primary site (static) manages
  U – set of available copies

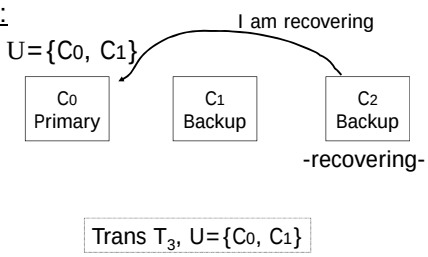
Update Transaction

(1) Get U from primary
(2) Get write locks from U nodes
(3) Commit at U nodes



A potential problem - example

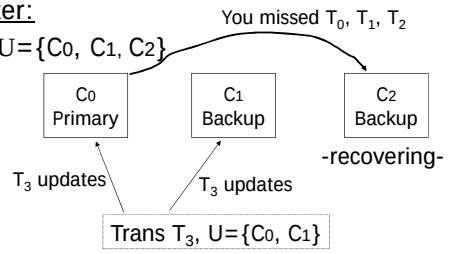
Now:



A potential problem - example

Later:

U={C0, C1, C2}



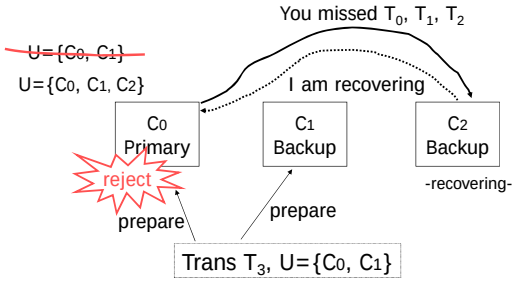
Solution:

- Initially transaction T gets copy of U' of U from primary (or uses cached value)
- At commit of T, check U' with current U at primary (if different, abort T)

Solution Continued

- When C_X recovers:
  - request missed and pending transactions from primary (primary updates U)
  - set write locks for pending transactions
- Primary polls nodes to detect failures (updates U)

Example Revisited



Available Copies — No Primary

- Let all nodes have a copy of U (not just primary)
- To modify U, run a special atomic transaction at all available sites (use commit protocol)
  - E.g.: $U_1=\{C_1, C_2\} \rightarrow U_2=\{C_1, C_2, C_3\}$ only C_1, C_2 participate in this transaction
  - E.g.: $U_2=\{C_1, C_2, C_3\} \rightarrow U_3=\{C_1, C_2\}$ only C_1, C_2 participate in this transaction

- Details are tricky...
- What if commit of U-change blocks?

Node Recovery (no primary)

- Get missed updates from any active node
- No unique sequence of transactions
- If all nodes fail, wait for { - all to recover / - majority to recover }

Example

recovering node

Committed: A,B,C,D,E,F Pending: G	Committed: A,B	Committed: A,C,B,E,D Pending: F,G,H

☛ How much information (update values) must be remembered? By whom?

Correctness with replicated data

X_1 X_2

S_1: $r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2]$

☞ Is this schedule serializable?

X_1 X_2

S_1: $r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2]$

☞ Is this schedule serializable?

One idea: Require transactions to update all copies

S_1: $r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2] \rightarrow w_1[X_2] \rightarrow w_2[X_1]$

X_1 X_2

S_1: $r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2]$

☞ Is this schedule serializable?

One idea: Require transactions to update all copies

S_1: $r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2] \rightarrow w_1[X_2] \rightarrow w_2[X_1]$

(not a good idea for high-availability algorithms)

Another idea: Build in copy-semantics into notion of serializability

One copy serializable (1SR)

A schedule S on replicated data is 1SR if it is equivalent to a serial history of the same transactions on a one-copy database

To check 1SR

- Take schedule
- Treat $r_i[X_j]$ as $r_i[X]$ $\quad X_j$ is copy of X
  $w_i[X_j]$ as $w_i[X]$
- Compute P(S)
- If P(S) acyclic, S is 1SR

Example

S_1: $r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2]$

S_1': $r_1[X] \rightarrow r_2[X] \rightarrow w_1[X] \rightarrow w_2[X]$

$T_2 \rightarrow T_1$

$T_1 \rightarrow T_2$

S_1 is <u>not</u> 1SR!

Second example

S_2: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2]$

$r_2[X_1] \rightarrow w_2[X_1] \rightarrow w_2[X_2]$

S_2': $r_1[X] \rightarrow w_1[X] \rightarrow w_1[X]$

$r_2[X] \rightarrow w_2[X] \rightarrow w_2[X]$

$P(S_2)$: $T_1 \rightarrow T_2$

S_2 is 1SR

Second example

S_2: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2]$

$r_2[X_1] \rightarrow w_2[X_1] \rightarrow w_2[X_2]$

S_2': $r_1[X] \rightarrow w_1[X] \rightarrow w_1[X]$

$r_2[X] \rightarrow w_2[X] \rightarrow w_2[X]$

- Equivalent serial schedule

  S_S: $r_1[X] \rightarrow w_1[X]$

  $r_2[X] \rightarrow w_2[X]$

Question: Is this a "good" schedule?

S_3: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2]$

$r_2[X_1] \rightarrow w_2[X_1]$

Question: Is this a "good" schedule?

S_3: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2]$

$r_2[X_1] \rightarrow w_2[X_1]$

S_3: $r_1[X] \rightarrow w_1[X] \rightarrow w_1[X]$

$r_2[X] \rightarrow w_2[X]$

S_3: $r_1[X] \rightarrow w_1[X]$

$r_2[X] \rightarrow w_2[X]$

to be valid schedule, need precedence edge between w1(X) and w2(X)

Question: Is this a "good" schedule?

We need to know how w2(X2) is resolved:

OK:

S_3: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2]$
$r_2[X_1] \rightarrow w_2[X_1]$
$w_2[X_2]$

Not OK:

S_3: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2]$
$r_2[X_1] \rightarrow w_2[X_1]$
$w_2[X_2]$

Question: Is this a "good" schedule?

S_3: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2]$
$r_2[X_1] \rightarrow w_2[X_1]$

Bottom Line: When $w_2[X_2]$ is missing because X_2 is down, assume X2 recover will perform $w_2[X_2]$ in correct order

S_3: $r_1[X] \rightarrow w_1[X]$
$r_2[X] \rightarrow w_2[X]$

Another example:

- S_3 continues with T3:

S_4: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \rightarrow r_3[X_2] \rightarrow w_3[X_1]$
$r_2[X_1] \rightarrow w_2[X_1]$
$w_3[X_2]$

Another example:

S_4: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \rightarrow r_3[X_2] \rightarrow w_3[X_1]$
$r_2[X_1] \rightarrow w_2[X_1]$
$w_3[X_2]$

S_4: $r_1[X] \rightarrow w_1[X] \rightarrow w_1[X] \rightarrow r_3[X] \rightarrow w_3[X]$
$r_2[X] \rightarrow w_2[X]$
$w_3[X]$

S_4: $r_1[X] \rightarrow w_1[X] \rightarrow r_3[X] \rightarrow w_3[X]$
$r_2[X] \rightarrow w_2[X]$

Seems OK but where do we place missing $w_2[X_2]$?

Another example:

$w_2[X_2]$ must be before $w_1[X_2]$ or after $r_3[x_2]$ (else t3 read different)

S_4: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \rightarrow r_3[X_2] \rightarrow w_3[X_1]$
$r_2[X_1] \rightarrow w_2[X_1]$
$w_3[X_2]$

Another example:

One option:

$w_2[X_2]$ must be before $w_1[X_2]$ or after $r_3[x_2]$ (else t3 read different)

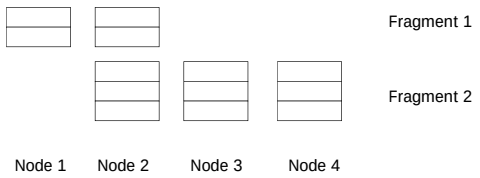
S_4: $r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \rightarrow r_3[X_2] \rightarrow w_3[X_1]$
$r_2[X_1] \rightarrow w_2[X_1]$
$w_3[X_2]$
$w_2[X_2]$

performed by X2 recovery

Outline

- Basic Algorithms
- Improved (Higher Availability) Algorithms
  - Mobile Primary
  - Available Copies (and 1SR)
- Multiple Fragments & Other Issues

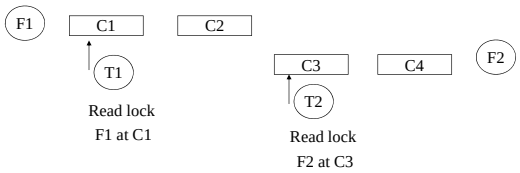
Multiple fragments



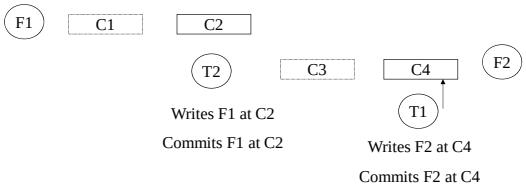
- A transaction spanning multiple fragments must
  - follow locking rules for each fragment
  - commit must involve "majority" in each fragment

- Careful with update transactions that read but do not modify a fragment

Example:



C1,C3 fail...



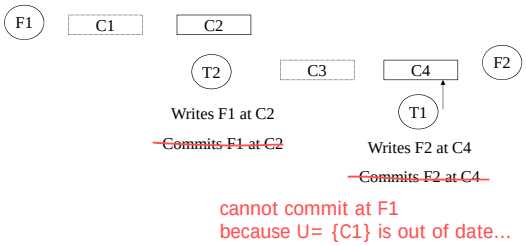
Equivalent history:

$r_1[F_1]$ $r_2[F_2]$ $w_1[F_2]$ $w_2[F_1]$

not serializable!

Solution: commit at read sites too

C1,C3 fail...



Read-Only Transactions

- Can provide "weaker correctness"
- Does not impact values stored in DB

	C1: primary	C2: backup
A:	0	0
B:	0	0

T1: A ← 3
T2: B ← 5

Later on:

T1: A ← 3
T2: B ← 5

	C1: primary		C2: backup	
A:	~~0~~ 3	B ← 5	~~0~~ 3	:A
B:	~~0~~ 5		0	:B

R1 read transaction sees current sate

R2 read transaction at backup sees "old" but "valid" state

States Are Equivalent

- States at Primary:
  - no transactions
  - T1
  - T1, T2
  - T1, T2, T3
- States at Backup:
  - no transactions
  - T1
  - T1, T2
  - T1, T2, T3

States Are Equivalent

- States at Primary:
  - no transactions
  - T1
  - T1, T2
  - T1, T2, T3
- States at Backup:
  - no transactions
  - T1
  - T1, T2
  - T1, T2, T3

at this point in time, backup may be behind…

Schedule is Serializable

- S1 = T1 R1 T2 T3 R2 T4 …(R1)…(R2)…

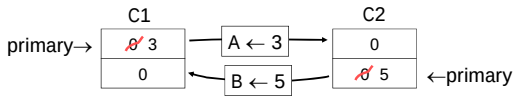
Example 2

- A, B have different primaries now
- 1-safe protocol used

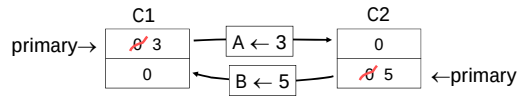
	C1		C2	
primary→ A:	0	:A	0	:A
B:	0		0	←primary

T1: A ← 3
T2: B ← 5

T1: A ← 3
T2: B ← 5



T1: A ← 3
T2: B ← 5



At this time:
- Q1: reads A, B at C1; sees T1 Q1 T2
- Q2: reads A, B at C2; sees T2 Q2 T1

Eventually:

- Schedule of update transactions is OK:
  - T1 T2 ≡ T2 T1
- Each R.O.T. sees OK schedule:
  - T1 Q1 T2 or T2 Q2 T1
- But there is NO single complete schedule that is "OK"…

- In many cases, such a scenario is OK
- Called weak serializability:
  - update schedule is serializable
  - R.O.T. see committed data

Data Replication

- RAWA, ROWA
- Primary copy
  - static [local commit or distributed commit]
  - mobile primary [2-safe (2PC or 3PC) or 1-safe]
- Available copies [with or without primary]
- Correctness (1SR)
- Multiple Fragments
- Read-Only Transactions

Issues

- To centralize control or not?
- How much availability?
- "Weak" reads OK?

Paxos

- Paxos is a replicated data management algorithm <u>supposedly</u> used in ZooKeeper.

Paxos for System Builders

Jonathan Kirsch and Yair Amir

Technical Report CNDS-2008-2 - March 2008
http://www.dsn.jhu.edu

Paxos with 2 Phases

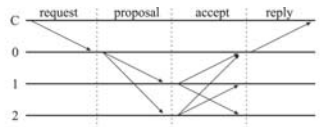


Figure 1: Paxos normal-case operation. Client C sends an update to the leader (Server 0). The leader sends a PROPOSAL containing the update to the other servers, which respond with an ACCEPT message. The client receives a reply after the update has been executed.

With these mechanisms in place, Paxos ensures agreement by introducing another round of communication – the *Prepare phase* – before the Proposal phase. In the Prepare phase, the leader asks the servers to respond with the latest proposal (if any) they have accepted.

Paxos with 2 Phases

- uses majority voting
- can handle partitions

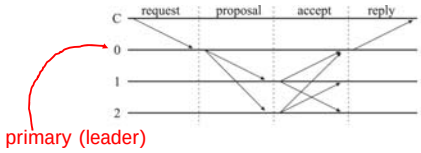


Figure 1: Paxos normal-case operation. Client C sends an update to the leader (Server 0). The leader sends a PROPOSAL containing the update to the other servers, which respond with an ACCEPT message. The client receives a reply after the update has been executed.

3 phase commit!

With these mechanisms in place, Paxos ensures agreement by introducing another round of communication – the *Prepare phase* – before the Proposal phase. In the Prepare phase, the leader asks the servers to respond with the latest proposal (if any) they have accepted.