Data Replication

- Reliable net, fail-stop nodes
- The model

<table>
<thead>
<tr>
<th>Database</th>
<th>node 1</th>
<th>node 2</th>
<th>node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fragment</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

• 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
  ![Diagram showing a locked X1]
  ![Diagram showing X2 and X3 with a cross]
  ![Diagram showing X cannot access X!]

Basic Solution — Improvement

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

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
  ![Diagram showing lock, write, commit, and propagate update]
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 \( t = 0 \)

<table>
<thead>
<tr>
<th>( X1 )</th>
<th>( X2 )</th>
<th>( X3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Z1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
T1: X \leftarrow 1; Y \leftarrow 1; \\
T2: Y \leftarrow 2; \\
T3: Z \leftarrow 3;
\]

Example \( t = 1 \)

\[
X1: 1 \\
Y1: 1 \\
Z1: 3
\]

\[
T1: X \leftarrow 1; Y \leftarrow 1; \quad \text{active at node 1} \\
T2: Y \leftarrow 2; \quad \text{waiting for lock at node 1} \\
T3: Z \leftarrow 3; \quad \text{active at node 1}
\]

Example \( t = 2 \)

\[
X1: 1 \\
Y1: 2 \\
Z1: 3
\]

\[
T1: X \leftarrow 1; Y \leftarrow 1; \quad \text{committed} \\
T2: Y \leftarrow 2; \quad \text{committed} \\
T3: Z \leftarrow 3; \quad \text{committed}
\]

What good is RPWP-LC?

- primary
- backup updates
- backup
- can’t read!
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

Example

X1: 0  X2: 0
Y1: 0  Y2: 0
Z1: 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

<table>
<thead>
<tr>
<th></th>
<th>Probability can read</th>
<th>Probability can write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROWA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPWP-LC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPWP-DC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison

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

<table>
<thead>
<tr>
<th>Probability</th>
<th>Read Prob.</th>
<th>Write Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>$P^n$</td>
<td>$P^n$</td>
</tr>
<tr>
<td>ROWA</td>
<td>$1 - (1-P)^n$</td>
<td>$P^n$</td>
</tr>
<tr>
<td>RPWP:LC</td>
<td>$P$</td>
<td>$P^n$</td>
</tr>
<tr>
<td>RPWP:DC</td>
<td>$P$</td>
<td>$P^n$</td>
</tr>
</tbody>
</table>

Comparison

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

<table>
<thead>
<tr>
<th></th>
<th>Read Prob.</th>
<th>Write Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.9510</td>
<td>0.9510</td>
</tr>
<tr>
<td>ROWA</td>
<td>1.0000</td>
<td>0.9510</td>
</tr>
<tr>
<td>RPWP:LC</td>
<td>0.9900</td>
<td>0.9900</td>
</tr>
<tr>
<td>RPWP:DC</td>
<td>0.9900</td>
<td>0.9510</td>
</tr>
</tbody>
</table>

Comparison

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

<table>
<thead>
<tr>
<th></th>
<th>Read Prob.</th>
<th>Write Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.3660</td>
<td>0.3660</td>
</tr>
<tr>
<td>ROWA</td>
<td>1.0000</td>
<td>0.3660</td>
</tr>
<tr>
<td>RPWP:LC</td>
<td>0.9900</td>
<td>0.9900</td>
</tr>
<tr>
<td>RPWP:DC</td>
<td>0.9900</td>
<td>0.3660</td>
</tr>
</tbody>
</table>

Comparison

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

<table>
<thead>
<tr>
<th></th>
<th>Read Prob.</th>
<th>Write Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.5905</td>
<td>0.5905</td>
</tr>
<tr>
<td>ROWA</td>
<td>1.0000</td>
<td>0.5905</td>
</tr>
<tr>
<td>RPWP:LC</td>
<td>0.9000</td>
<td>0.9000</td>
</tr>
<tr>
<td>RPWP:DC</td>
<td>0.9000</td>
<td>0.5905</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>primary</th>
<th>new primary</th>
<th>backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>committed: T1, T2</td>
<td>need to get and apply: T1, T2</td>
<td></td>
</tr>
</tbody>
</table>

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

(3) Resolve pending transactions

<table>
<thead>
<tr>
<th>primary</th>
<th>new primary</th>
<th>backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3?</td>
<td>T3 in “W” state</td>
<td>T3 in “W” state</td>
</tr>
</tbody>
</table>

Failed Nodes: Example

<table>
<thead>
<tr>
<th>now:</th>
<th>primary</th>
<th>backup</th>
<th>backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>-down-</td>
<td>commit T3</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>-down-</td>
<td>P3</td>
<td>-down-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>later:</th>
<th>primary</th>
<th>backup</th>
<th>backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>-down-</td>
<td>P2</td>
<td>-down-</td>
</tr>
<tr>
<td>P3</td>
<td>-down-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>later still:</th>
<th>primary</th>
<th>backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>-down-</td>
<td>commit T2 (unaware of T1)</td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

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

Example: Majority Commit

C1: T1, T2, T3
C2: T1, T2
C3: T1, T3

Example: Majority Commit

t1: t1: C1 fails
t2: t2: C2 new primary
t3: t3: C2 commits T1, T2, T3; aborts T4
t4: t4: C2 resumes processing
t5: t5: C2 commits T5, T6
t6: t6: C1 recovers; asks C2 for latest state
t7: 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):

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:

<table>
<thead>
<tr>
<th>primary</th>
<th>backup 1</th>
<th>backup 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2, T3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

later:

<table>
<thead>
<tr>
<th>primary</th>
<th>next primary</th>
<th>backup 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2, T3, T4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

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

**Problem: Lost Transactions**

now:

<table>
<thead>
<tr>
<th>primary</th>
<th>backup 1</th>
<th>backup 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2, T3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

later:

<table>
<thead>
<tr>
<th>primary</th>
<th>next primary</th>
<th>backup 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T4, T5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>primary</th>
<th>next primary</th>
<th>backup 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2, T3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

later:

<table>
<thead>
<tr>
<th>backup 3</th>
<th>next primary</th>
<th>backup 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2, T3, T4, T5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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:

U = {C₀, C₁}

I am recovering

Trans T₃, U = {C₀, C₁}
A potential problem - example

Later:
\[ U = \{C_0, C_1, C_2\} \]

- T_3 updates \( C_0 \)
- T_3 updates \( C_1 \)

You missed T_0, T_1, T_2

Trans T_3, U = \{C_0, C_1\}

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 majority to recover

Example
recovering node

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

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

Correctness with replicated data

S1: r1[X1] → r2[X2] → w1[X1] → w2[X2]

Is this schedule serializable?

One idea: Require transactions to update all copies
S1: r1[X1] → r2[X2] → w1[X1] → w2[X2] → w1[X2] → w2[X1]

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 \( r_i[X] \) as \( r_i[X] \) \( X_j \) is copy of \( X \)
- \( w_i[X] \) 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 not 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

\[ S_5: r_1[X] \rightarrow w_1[X] \rightarrow w_2[X] \]

**Question: Is this a “good” schedule?**

\( S_3: r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \rightarrow w_2[X_1] \)

**Question: Is this a “good” schedule?**

\( S_3: r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \rightarrow w_2[X_1] \)

\[ r_2[X_1] \rightarrow w_2[X_1] \rightarrow w_2[X] \]

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

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

\[ T_2 \rightarrow T_1 \rightarrow T_2 \]

\( T_1 \rightarrow T_2 \)

\( S_3: r_1[X] \rightarrow w_1[X] \rightarrow w_2[X] \rightarrow w_2[X] \)

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

\( S_3: r_1[X] \rightarrow w_1[X] \rightarrow w_2[X] \rightarrow w_2[X] \)

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

\( S_3: r_1[X] \rightarrow w_1[X] \rightarrow w_2[X] \rightarrow w_2[X] \)

\[ r_2[X] \rightarrow w_2[X] \rightarrow w_2[X] \]
Question: Is this a “good” schedule?

We need to know how \( w_2(X_2) \) is resolved:

**OK:**

\[
\qquad r_2[1] \rightarrow w_2[1] \\
\]

**Not OK:**

\[
\qquad r_2[1] \rightarrow w_2[1] \\
\]

We need to know how \( w_2(X_2) \) is resolved:

**OK:**

\[
\qquad r_2[1] \rightarrow w_2[1] \\
\]

**Not OK:**

\[
\qquad r_2[1] \rightarrow w_2[1] \\
\]

**Bottom Line:** When \( w_2(X_2) \) is missing because \( X_2 \) is down, assume \( X_2 \) recover will perform \( w_2(X_2) \) in correct order.

**Another example:**

- \( S_3 \) continues with \( T_3 \):

\[
\qquad r_2[1] \rightarrow w_2[1] \\
\]

**Seems OK but where do we place missing \( w_2[2] \)?**

**Another example:**

\[
\quad r_1[1] \rightarrow w_2[1] \\
\quad r_3[1] \rightarrow w_3[2] \\
\]

**Another example:**

- \( S_4 \) must be before \( w_1(X_2) \) or after \( r_3[2] \) (else \( T_3 \) read different)

\[
\quad r_2[1] \rightarrow w_2[1] \\
\]

**One option:**

- \( w_2[2] \) must be before \( w_1[2] \) or after \( r_3[2] \) (else \( T_3 \) read different)

\[
\quad r_2[1] \rightarrow w_2[1] \\
\]

**performed by \( X_2 \) recovery**

**Another example:**

\[
\quad r_2[1] \rightarrow w_2[1] \\
\]
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:

Equivalent history:

\[ r_1[F_1] \quad r_2[F_2] \quad w_1[F_2] \quad w_2[F_1] \]

not serializable!

Solution: commit at read sites too
Read-Only Transactions

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

C1: primary
A: 0
B: 0

C2: backup
A: 0
B: 0

T1: A ← 3
T2: B ← 5

Later on:

C1: primary
A: 0
B: 0

C2: backup
A: 0
B: 0

R1 read transaction
sees current state

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
primary
A: 0
B: 0

C2
A: 0
B: 0

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 supposedly used in ZooKeeper.

Paxos for System Builders
Jonathan Kirsh and Yair Amir
http://wwwdsn.jhu.edu

Paxos with 2 Phases

- uses majority voting
- can handle partitions