Distributed Operating Systems, Part III

CS 239, Spring 1995

May 9, 1995

Outline of Introductory Materials

- Why distributed operating systems?
- Important issues in distributed OS's
- Important distributed OS tools and mechanisms

Important Distributed Operating System Tools and Mechanisms

- Caching and replication
- Transactions and two-phase commit
- Hierarchical name spaces
- Optimistic methods

What Problems Could Arise For Distributed Transactions?

- Other processes could write the variables
- Other processes could read the variables
- Failures could interrupt the process

How can we avoid these problems?

Requirements for Atomicity

An action is atomic if

- Process controlling it is unaware of activities of other processes and visa versa
- No communications with other processes
- No outside detectable state changes
- Indivisible and instantaneous to the rest of the system

Commit and Abort

A *commit* is an unconditional guarantee that a transaction will be completed

An *abort* is an unconditional guarantee that a transaction will be completely backed out

In both cases, regardless of multiple failures

All transactions eventually either commit or abort

Aborting Transactions

Requires returning system to its pre-transaction state

Essentially a rollback operation

Typically done either by logging changes

- Or delaying updates until commit point is reached

Reasons for Aborting

- Deadlocks
- Timeouts
- Protection violations
- Consistency violations
- Site or network failures

Abort Mechanisms

- Write-ahead logs
- Shadow pages

These techniques generally useful for recovery from failures

Write-Ahead Logs

An operation-based abort mechanism

- Record operations and undo them, if necessary

For each operation performed, record enough information to completely undo it

- Either old state or other information

Undo log

Write-Ahead Log Protocol

- Write the undo log to stable storage
- Make the update
- If transaction commits, undo log can be deleted/garbage collected
- If transaction aborts, use undo log to roll back operation
  - And then delete/garbage collect the log

Pros/Cons of Write-Ahead Logs

+ Only changed data need be logged

– More complex garbage collection

– Hard to provide concurrent view of data to another process

– Requires save on every data item written

Shadow Pages

State-based approach

Save a checkpoint of the state before performing transaction operations

If transaction aborts, restore old state

Can be expensive if state is large

Shadow paging limits the cost

Shadow Page Algorithm

Before writing a data item in a transaction,

1. Make a complete copy of its page
2. Allow transaction to use the copy (transparently)
3. On commit, switch shadow page for new page
4. On abort, restore shadow page

Pros/Cons of Shadow Paging

+ Can provide multiple concurrent views of object
+ Simple garbage collection
– Can only save at "page" granularity
– Requires save on every page written

Distributed Commitment Mechanisms

Mechanisms to guarantee atomicity of actions in a distributed system

Important mechanisms for building distributed transactions

Works with mechanisms to permit backing out of transactions

The Generals Problem

An abstraction of the commit problem

A group of independent generals must cooperate to attack a fortress

If all attack at once, they will succeed

But if any do not attack, they will fail

Communications only by messengers

Who can be delayed or captured

The Generals Problem, Con't

If all the generals wish to cooperate, how can they reach agreement on when to attack under these conditions?

Cannot reach agreement in a fixed number of messages

But, if messengers eventually get through, can be done in an indeterminate number of msgs

Generals and Transactions

Transactions are like the attack

The generals are like the sites participating in the attack

The attack time is like the commit point

How do the sites in a transaction agree it's OK to commit?

Two-Phase Commit

Algorithm to allow sites in a transaction to agree to commit

Or, equivalently, agree to abort

One process/site is the *coordinator*

The others are the *cohorts*

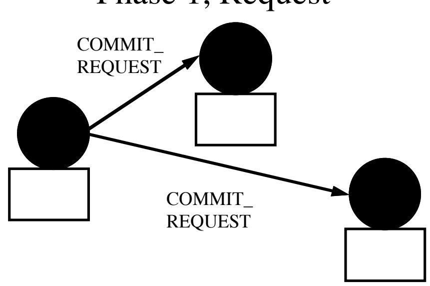
Two-Phase Commit Algorithm

- Phase 1 -
  - Coordinator sends COMMIT-REQUEST to cohorts
  - Coordinator waits for all replies
  - Each cohort responds with a AGREED or ABORT message (after writing local transaction information to stable storage)

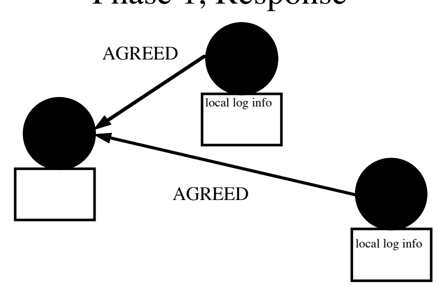
Two-Phase Commit Algorithm, Con't

- Phase 2
  - If all cohorts agree, coordinator writes COMMIT record to his log and sends COMMIT message to all cohorts (else ABORT)
  - Coordinator waits for all acks
  - Cohorts take appropriate action and acknowledge
  - Coordinator writes COMPLETE to its log

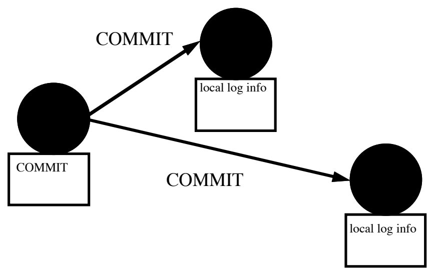
Two-Phase Commit Diagram Phase 1, Request



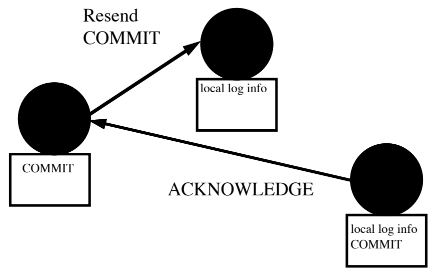
Two-Phase Commit Diagram, Phase 1, Response



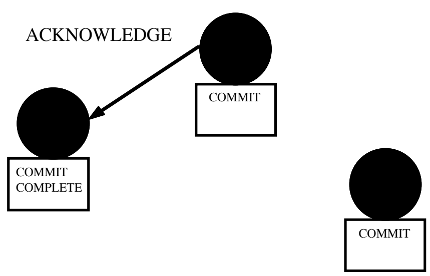
Two-Phase Commit Diagram, Phase 2, Request



Two-Phase Commit Diagram, Phase 2, Waiting



Two-Phase Commit Diagram, Phase 2, Response



Characteristics of Two-Phase Commit

Nobody commits until everyone agrees to

Timeouts handle lost/delayed messages

Examination of logs of participants in cases of failures allow recovery

Members block while waiting for completion

Two-Phase Commit - Recovering From Failure

- Coordinator fails before writing COMMIT record to log
  - On recovery, broadcast ABORT
- Coordinator fails between COMMIT and COMPLETE
  - On recovery, broadcast COMMIT

Two-Phase Commit - Recovering From Failure, Con't

- Coordinator fails after writing COMPLETE
  - Transaction succeeded, do nothing
- Cohort crashes in Phase 1
  - Coordinator aborts transaction
- Cohort crashes in Phase 2
  - On recovery, check with coordinator whether to commit or abort

Naming in Distributed Systems

Processes can only use resources that they can name

System must be able to translate names into locations

Even for objects it's never heard of before

Naming Terms

- A *name* specifies what a process seeks
- An *address* specifies where the object is
- A *route* specifies how to get there

Names are mapped onto addresses

Addresses are mapped onto routes

Names and Name Spaces

Names have meaning within specific *name spaces*

All names in a name space comply to the naming rules of that space

Name Space Structures

- Flat
- Partitioned
- Descriptive

Hierarchical name spaces are a special case of partitioned name spaces

Hierarchical Name Spaces

Name space is organized into a number of levels

Rooted at some point

Examples -

Unix file name space

Internet site names

Advantages of Hierarchical Name Spaces

- Built-in organization
- Easy compartmentalization of name space
- Handles arbitrary scale well
- Mathematically well understood
  - Maps directly to a tree
- Generally understood by people

Names in Distributed Systems

Entire namespace typically not stored locally

So there must be a method to find objects with names in remote parts of name space

How do you get from your part of the name space to another machine's part?

Approach 1 - Explicit Site Names

Name contain the site name of the site that handles them

- E.g., /wells.cs.ucla.edu/users/reiher

To find named object, extract site from the name and send request to that site

Typically some shorthand for local names

Pros/Cons of Explicit Site Names

+ Easy to implement
+ Easy to find remote resources

– Poor location transparency

- Network is not concealed
- Objects change names when they change machines

Approach 2 - Local Mount Points

Splice portion of remote name space into the local name space

Keep records to help traverse naming attempts across the splice

When necessary, transfer name resolution requests to remote site

Pros/Cons of Local Mount Points

+ Hides the network from ordinary users
+ Largely location transparent
  - Files don't change names when they move
  - Location not explicit in name

– Implementation more complicated

– Not name transparent

– Heavy setup costs

Approach 3 - Global Namespaces

All names exist in a single global namespace

System software translates names that cross machine boundaries

Often, in practice, explicit site names creep in

- But different in meaning than in Approach 1

Pros/Cons of Global Name Spaces

+ Fully name and location transparent
– Potentially expensive to implement and maintain
– Potential for poor scaling
– Requires agreement among all sites involved

A matter of much philosophical debate

Translating Names

Names of local objects typically translated locally

Mechanisms for translating remote names:

- Broadcast
- Forward to storing site
- Name servers

Optimistic Methods

In many cases, being almost certain of being right is much cheaper than being absolutely certain

In such cases, optimistic methods may pay off

Assume it's OK to proceed

Clean up the mess afterwards if you're wrong

Tradeoffs in Optimistic Methods

Optimism pays off if

The benefit is high enough

The chance of being wrong is low enough

The cost of being sure is high enough

And the cost of cleaning up mistakes is low enough

Particularly well suited to distributed systems

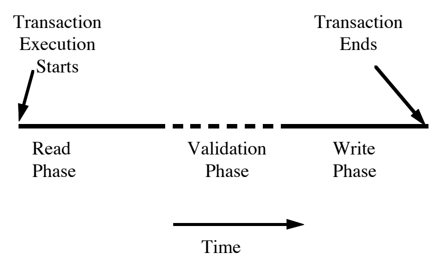
Optimistic Transaction Mechanisms

Whenever a transaction may be in synchronization difficulties, proceed as if it weren't

When transaction completes, check to see if synchronization problems really arose

Roll back transaction if they did

Optimistic Transaction Diagram

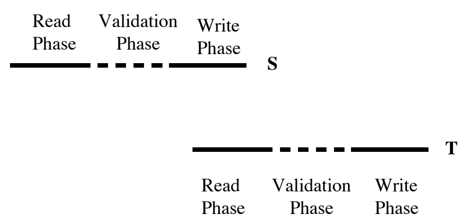


Validating Optimistic Transactions

During validation phase of transaction **T**, check if some other transaction -

1. The write phase of another transaction **S** intersects the read phase of **T**
2. The readset of **T** and the writeset of **S** intersect

Conflicting Transactions



Assuming intersection of readset(**T**) and writeset(**S**)

Handling Conflicting Transactions

If these conditions occur, roll back **T**

Since **T** is a transaction, you have to be able to roll it back, anyway

Restart **T** and try again

Potential for fairness problems with very long transactions

Optimistic Data Replication

Conservative methods of data replication guarantee no concurrent updates

E.g., voting and token schemes

Always at the expense of availability

Optimistic data replication methods give full availability

At the expense of possible concurrent updates

How Optimistic Data Replication Works

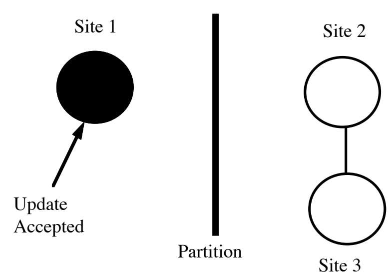
If a replica of the data is available, normal updates are permitted

Regardless of state of other replicas

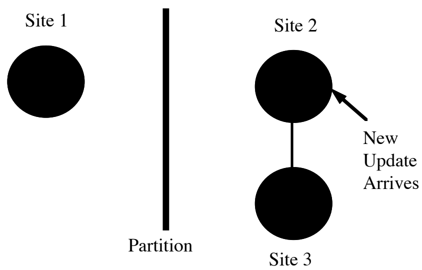
If problems arise, detect them later

And do something

Optimistic Replication Diagram



Concurrent Update Diagram



Problems With Optimistic Replication

- Propagating updates to partitioned sites
  Potentially harder than in conservative cases
- Detecting concurrent updates
- Handling concurrent updates

Propagating Updates

Rely on normal update propagation to get updates to non-partitioned sites

Periodically run another algorithm to detect and propagate changes to sites that missed them

Detecting Concurrent Updates

- Log based methods
  Requires keeping a lot of unnecessary data
  But may help deal with concurrent updates
- Version vectors
  Compact encoding of update information
  Helps detect updated files
  Doesn't help fix concurrent updates

Fixing Concurrent Updates

- Rollback mechanisms
- Versioning schemes
- Semantics-based conflict resolution
- User intervention

Critique of Optimistic Data Replication

+ High availability
+ Works well if shared write data uncommon
+ No distributed algorithms on normal access
– Can be troublesome if partitions and shared write data both common
– May require excessive user intervention
– Permits using "bad" data