Distributed File System

• Last Class
  – NFS
  – Design choices
  – Opportunistic locking
  – Local name spaces
DFS Basic Info

- Global name space across all servers in DFS (or Cell in DCE speak)
- Hierarchy Is partition into filesets
  - Stored at a different servers
- Mapping stored in fileset location database (FLDB)
DFS Mount Points

FileSets are given names independent of hierarchy

Mapping of FileSet name to server stored in FLDB

mounting = symbolic link to the name

To use FS must use name to look up server in FLDB

Provides global name space: all clients use same name.
DCE DFS V. NFS

• DCE:
  – Single global name space
  – Namespace is build into the central db (directory hierarchy)
  – Mounts/lookups must use central databases

• NSF
  – Independent disjoin names spaces based on mount
Strict Consistency in DFS

- **FS**
  - **System**
    - **sol7**
      - **bin**
    - **osf1**
      - **bin**
      - **usr**
    - **twd**
    - **motif**
    - **dce**
  - **Users**
  - **Projects**

To read/write files.

Clients request tokens.

Server provides token and allows caching.

As long as tokens are held, client can make appropriate changes.
DFS Tokens (1)

File A: Read:0-4095
File B: Write:0-512

File A: Read:0-4095
File B: Write:513-4096
DFS Tokens (2)

```
sol7
  |---- bin
  |    |---- bin

osf1
  |---- bin
  |    |---- usr

  |---- twd

motif

  |---- dce

```

```
Client

File A: Read:0-4095

Write(A, 0-4095)

File A: Read:0-4095

Client

Server

Server

Server

Server

FLDB

FLDB

FLDB

FLDB
```
DFS Tokens (3)

Client

File A: Read:0-4095

File A: Read:0-4095

Revoke(A, Read, 0-4095)
DFS Tokens (4)

Client

File A: Read: 0-4095

File A: Write: 0-4095

Server

Grant(A, write, 0-4095)
DFS Tokens (5)

Client

File A: Read:0-4095
File A: Write:0-4095

Server

Read(A, 0-4095)
DFS Tokens (6)

Client

File A: Read:0-4095

File A: Write:0-4095

Server

Revoke(A, write, 0-4095)
DFS Tokens (7)

Client

File A: Read:0-4095

Grant(A, read, 0-4095)

File A: Read:0-4095

Server

Server

Server

Server

FLDB

FLDB

FLDB
Consistency Models

- **NFS**: weak or strict
  - Strict if locks are used

- **CIFS**: strict consistency
  - Uses opportunistic locks
  - Revoke + flush

- **DCE**: Strict consistency
  - Uses tokens
  - Similar revoke+flush policy as CIFS
Crash Recovery
DFS Crash Recovery

Client crash: server reclaims token
- Server should be able to revoke when client down

Server crash: on reboot rebuilt token DB
- Client should keep using cache even when server failed

Network failure: both are up but can’t communicate
- Revoking tokens or using cache create issues
- Compromise:
  - Client uses token until expire
  - Server revokes unilateral
  - When failure fixed, client changes may be rejected!!!
DFS Recovery Problems

• Client application must participate!
  – must recognize that operation returns “timed-out” error
  – must retry

• Due to semantics of tokens, it isn’t feasible to provide NFS-style hard mount.
  – Hard mount possible because NFS is stateless
  – In DCE DFS state complicates hard mounting
Server State

• It’s required!
  – exact Unix semantics
    - E.g. Need to know how many clients using file to delete
  – mandatory locks
State Recovery

• Recovery from crashed server
  – clients reclaim state on server
    - grace period after crash during which no new state may be established

• recovery from crashed client
  – server detects crash and nullifies client state information on server
Coping with Non-Responsiveness

• Leases
  – locks are granted for a fixed period of time
    - server-specified lease
  – if lease not renewed before expiration, server may (unilaterally) revoke locks and share reservations
    - most client RPCs renew leases
  – clients must contact server periodically
    - if clientid is rejected as stale, then server has restarted
    - server’s grace period is equal to lease period
DFS ≠ LFS

• Distributed file system != Local file system

• Servers might give up on non-crashed clients
  – clients may lose locks
  – clients may lose files
  – NFSv4 attempts to make such things unlikely
Distributed File System
Summary

• Opportunistic Locking
  – Enables client to safely cache file and modify it
  – Improves performance

• Global V. Local name spaces
  – Global: DCE DFS
  – Local: NFS

• Crash recovery
  – Requires client participation
  – Leases help deal with non responsiveness
  – DFS != LFS
CS 138: Distributed Transactions
Today

• Transactions
  – Atomic Commit of Transactions
  – Five desired properties

• Two Phase Commit
  – Analyzed state machines for different roles
  – Participant and Coordinator response to failures
  – Blocking issue

• Three Phase Commit
  – Attempt to eliminate Blocking issue
  – More complex protocol including more messages
  – Analyzed state machines for different roles

• Much of this lecture is adapted from the textbook by Tanenbaum and Van Steen and from Chapter 7 of *Concurrency Control and Recovery in Database Systems*, by P. Bernstein, V. Hadzilacos, and N. Goodman, Addison-Wesley (1987).
Transactions

• “ACID” property:
  – atomic
    - all or nothing
  – consistent
    - take system from one consistent state to another
  – isolated
    - have no effect on other transactions until committed
  – durable
    - persists
Recall: Configuration Changes in Raft!

Cannot switch directly from one configuration to another: conflicting majorities could arise

See the paper for details
Roles in Atomic Transactions

• C = Coordinator
  – Initiates a change

• P = Participants
  – All nodes involved in making the atomic change
Distributed Transactions

Client (Participant)

Begin Transaction;
   a. withdraw(100);
   b. deposit(50);
   c. deposit(50);
End Transaction;

A (Participant)
withdraw(100);

B (Participant)
deposit(50);

C (Participant)
deposit(50);
Coordination

Begin Transaction;
a. withdraw(100);
b. deposit(50);
c. deposit(50);
End Transaction;

Coordinator

withdraw(100);

A
(Participant)

deposit(50);

B
(Participant)

deposit(50);

C
(Participant)
Atomic Commit Properties

• AC1: All participants that reach a (commit/abort) decision reach the same one

• AC2: A participant cannot reverse its decision

• AC3: The commit decision can be reached only if all participants agree

• AC4: If there are no failures and all participants vote yes, then decision will be commit

• AC5: For any execution, if all failures are repaired and no new failures occur for a sufficiently long interval, then all participants will reach a (commit/abort) decision
Two-Phase Commit

• Phase 1
  – coordinator prepares to commit:
    - asks participants to vote either “commit” or “abort”
    – participants respond appropriately

• Phase 2
  – coordinator decides outcome:
    - if all participants vote commit, outcome is commit, otherwise outcome is abort
    - outcome sent to all participants
    – participants do what they’re told
Two-Phase Commit

Coordinator

pariticpants

Make a change

Commit or Abort

If all Commit
Then Commit
If any Abort
Then Abort

Response

Decide to abort
Or commit based
On local state

Once a node responds.
The node can’t change
responses

Commit or Abort
Two-Phase Commit

Coordinator

Make a change

Commit or Abort

If all Commit
Then Commit
If any Abort
Then Abort
(AC3, AC4)

Response

Decide to abort
Or commit based
On local state

Once a node responds.
The node can’t change
responses (AC2)

Commit or Abort (AC1)
Coordinator State Diagram for Two-Phase Commit

- **Init**: app commit/vote req
- **Wait**: any abort/abort, all commit/commit
- **Abort**: Coordinator
- **Commit**: Coordinator
- **Response**: Make a change
- **Commit or Abort**: Coordinator

Participants
Participant State Diagram for Two-Phase Commit

Coordinator

Make a change

Commit or Abort

Response

Vote req/abort

Vote req/commit

Abort

Commit

Uncertain

Init

Abort/ack

Commit/ack

Participant
State Diagrams for Two Phase Commit

Coordinator

Init → Wait
- app commit/vote req
- any abort/abort
- all commit/commit

Wait → Abort
- Commit

Participant

Init → Uncertain
- vote req/commit
- vote req/abort
- abort/ack
- commit/ack

Uncertain → Abort
- Commit
Failures

• Coordinator or participants could crash
  – assume “fail-stop”
    - crash detected by time-out
    - no byzantine failures
  – crashed machines restart
    - recover their state

• Focus on two failures
  – Node failures
  – Network failures (i.e., Lost messages)
Crash Points

Coordinator

- **Init**
  - app commit/vote req
  - any abort/abort
  - all commit/commit

- **Wait**
  - Abort
  - Commit

- **Abort**
- **Commit**

Participant

- **Init**
  - vote req/commit
  - abort/ack
  - commit/ack

- **Uncertain**
  - vote req/abort

- **Abort**
- **Commit**
Crash Points

Coordinator

Wait
- Init
  - app commit/vote req
  - any abort/abort
  - all commit/commit

Abort
Commit

Participant

Uncertain
- Init
  - vote req/abort
  - vote req/commit
  - abort/ack
  - commit/ack

Abort
Commit
Coordinator Recovery from Participant Failure

- Coordinator in wait state
  - It detects failure of participant
    - Using timeout
    - waiting for participant to say “commit” or “abort”

- Coordinator in *Wait* state
  - can’t assume either outcome
  - Waiting for them to respond
  - Takes no response as an abort
  - Abort transaction!!

Diagram:

- **Init**
  - app commit/vote req
  - any abort/abort
  - all commit/commit

- **Wait**
  - Abort
  - Commit

Coordinator
Participant Recovery from Coordinator Failure

- Participant in Uncertain state
  - It detects failure of coordinator
    - Using timeout
    - waiting for coordinator to say “commit” or “abort”
- Participant in Uncertain state
  - can’t assume either outcome
  - waits for coordinator to restart
    - On restart contact coordinator for final outcome
Optimizing Participant Recovery

• Waiting for Coordinator to restart can take a LONG time!
  – Especially if coordinator takes a while to reboot

• Participants contact other participants
  – p contacts q (p is in Uncertain state)
  – q is in:
    - commit (or abort) state
      • p goes to commit (or abort) state
    - init state (hasn’t yet voted)
      • both q and p abort
    - uncertain state
      • both p and q remain uncertain
Optimizing Participant Recovery

- q is in:
  - commit (or abort) state
    - p goes to commit (or abort) state
  - init state (hasn’t yet voted)
    - both q and p abort
  - uncertain state
    - both p and q remain uncertain
Optimizing Participant Recovery

- q is in:
  - commit (or abort) state
    - p goes to commit (or abort) state
  - init state (hasn’t yet voted)
    - both q and p abort
  - uncertain state
    - both p and q remain uncertain
Optimizing Participant Recovery

- q is in:
  - *commit* (or *abort*) state
    - p goes to *commit* (or *abort*) state
  - *init* state (hasn’t yet voted)
    - both q and p abort
  - *uncertain* state
    - both p and q remain *uncertain*

When all active nodes are uncertain, then this optimization does not provide a benefit.
Two-Phase Commit

- Works fine in practice!

- But …
  - all participants could conceivably be in uncertain state and coordinator is down (for a long time)

- Can we make it so such blocking can’t happen?
What Causes Blocking?

• Coordinator is down

• If all operational (not-failed) participants are in uncertain state, they are blocked

• If all participants are operational, they can elect new coordinator

• If any participant has crashed, the others don’t know if it crashed before or after voting (to commit or abort)
Guaranteeing Non-Blocking

- Non-blocking property (NB):
  - if any operational process is in the Uncertain state, then no process (operational or failed) can have decided to commit
- If NB holds, then operational processes may elect new coordinator and decide to commit or abort
Failures

• Coordinator or participants could crash
  – no communication failures
  – assume “fail-stop”
    - crash detected by time-out
    - no byzantine failures
  – crashed machines restart
    - recover their state
Three-Phase Commit

• Phase 1
  – coordinator prepares to commit:
    - asks participants to vote either “commit” or “abort”
    – participants respond appropriately

• Phase 2
  – coordinator counts votes:
    - if all participants vote commit, outcome is pre-commit, otherwise outcome is abort
    - outcome sent to all participants
    – participants ack and either abort or wait for commit

• Phase 3
  – coordinator waits for all acks
    - if committing, sends final commit to all participants
    – participants commit
Two-Phase Commit

Coordinator

If all Commit
Then Commit
If any Abort
Then Abort

Commit or Abort

Pre-commit/Abort

ACK

Commit

Make a change

Decide to abort
Or commit based
On local state

Pre-Commit or Abort
Revised State Diagrams

Init

app commit/vote req

Wait

any abort/abort

all commit/precommit

Abort

Pre Commit

Commit

vote req/abort

Abort

Pre Commit

Commit

vote req/commit

Uncertain

abort/ack

precommit/ack

Commit

Commit

all ack/commit

commit/commit

Commit
Revised State Diagrams

Init → Wait
- app commit/vote req
  - any abort/abort
    - Abort
  - all commit/precommit
    - Pre Commit
      - all ack/commit
        - Commit
    - Uncertain
      - abort/ack
      - precommit/ack
        - Pre Commit
          - commit/commit
            - Commit
Coordinator Recovery from Participant Failure

- Coordinator in *Wait* state
- It detects failure of a participant
  - Using timeout
  - *waiting for participant* to say “commit” or “abort”

- Coordinator in *Wait* state
  - can’t assume either outcome
  - Waiting for them to respond
  - Interprets absence of participant response as an abort
  - Abort transaction!!

- Same as 2PC
Coordinator Recovery from Participant Failure

- Coordinator in PreCommit state
  - It detects failure of participant
    - Using timeout
    - waiting for participant to say ACK

- Coordinator in Pre-Commit state
  - CAN assume they will commit
  - So commit independently
  - Failed nodes will learn on reboot
    - Either from coordinator or friend
Participant Recovery from Coordinator Failure

- Participant in init state
- It detects failure of coordinator
  - Using timeout
  - waiting for coordinator to propose a vote
- Safely Abort without waiting
Participant Recovery from Coordinator Failure

- Participant in uncertain state
- It detects failure of coordinator
  - Using timeout
  - waiting for coordinator to say “commit” or “abort”

- Participant in *Uncertain* state
  - can’t assume either outcome
  - Contact another node to determine status
Participant Recovery from Coordinator Failure

- Participant in Precommit state
- It detects failure of coordinator
  - Using timeout
  - waiting for coordinator to say “commit”

- Participant, p, in PreCommit state
  - Why not just COMMIT?
  - There might be a node, q, out there in uncertain state
  - If p commits and fails, there’s a chance q can still ABORT and violate the earlier assumptions
Details (1)

• If original coordinator remains operational
  – participant crashes handled as in two-phase commit

• If participant times out in *Uncertain state*
  – if any other participant has aborted, it aborts
  – otherwise, it starts an election for a new coordinator
    - All in Uncertain or some in uncertain, some in Precommit

• If participant times out in *PreCommit* state
  – it starts an election for a new coordinator
  – All in precommit or some in uncertain, some in Precommit
  – *No node can be in the Abort state if any node is in PreCommit.*
  – *If any in Commit then they move into Commit*
Participant Times out In Uncertain State

- Participant in uncertain state
  - waiting for coordinator to say “commit” or “abort”

- Other participants can be in:
  - Uncertain - these participants responded to vote with Yes
  - Abort - these participants responded to vote with Abrot
  - Init - these participants have not received the request to vote
  - NO participant can be in PreCommit
Participant Times out In PreCommit State

- Participant in Precommit state
  - waiting for coordinator to say “commit”

- Other participants can be in:
  - Precommit – received pre-commit msg from coordinator but waiting for commit
  - Commit – received commit
  - Uncertain – these participants have not received “precommit” from coordinator.
    - Coordinator could have failed before sending precommit to them
  - NO participant can be in Init/Abort
    - All must have voted yet for coordinator to send PreCommit
Coordinator Recovery: New Coordinator

• When newly elected coordinator starts up
  – sends state-request message to all operational participants

• Coordinator collects states and proceeds according to four termination rules (termination protocol):
  – TR1: if any participant is in Abort state, all are sent abort messages
  – TR2: if some participant is in Commit state, all are sent commit messages
  – TR3: if all participants are in Uncertain state, all are sent abort messages
  – TR4: if some participant is in PreCommit state, but none in Commit State, those in Uncertain state are sent PreCommit messages; once these are acked, all participants are sent commit messages
Total Failure

• What if coordinator and all participants fail?

• When they come back up, how do they decide?
  – if resurrected participant either didn’t vote or voted abort, it may unilaterally abort
    - otherwise, must run termination protocol (from last slide)

  – but works only if last participant to fail has come back up
    - If last participant doesn’t come back up, you don’t know its current state
Communication Failures

• Network could partition into multiple pieces
• Not sufficient to get agreement in a piece containing a quorum
  – consensus is required for commit!

• Scenario
  – all participants vote
  – coordinator collects results
  – network partitions before or after all results collected
  – if network reconnects: easy
  – network never fully reconnects, but each participant eventually can communicate (perhaps briefly) with all others
Summary of Distributed Transaction

• Transactions
  – Atomic Commit of Transactions
  – Five desired properties

• Two Phase Commit
  – Analyzed state machines for different roles
  – Participant and Coordinator response to failures
  – Blocking issue

• Three Phase Commit
  – Attempt to eliminate Blocking issue
  – More complex protocol including more messages
  – Analyzed state machines for different roles