CSci 5105

Introduction to Distributed Systems

Byzantine, Recovery
Last Time

• Fault tolerance
• Reliable multicast
Today

• FT continued
• Recovery
Two-Phase Commit (2PC)

• General protocol to implement reliable multicast and forms of consensus
• Send message and have everyone either act on message or not
• Typical action: commit a transaction
• Multi-step (with coordinator)
  – Vote-request
  – Vote-commit or vote-abort
  – Global-commit or global-abort
Two-Phase Commit (2PC)

- Distributed commit - all or none
- Starts when someone wants to commit a value and asks coordinator if it is ok
What about failure?

- Coordinator failure
- Node P in READY state and times out
- Asks node Q

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Must block if everyone in READY state
2PC Failure/Recovery

- Nodes fail and may recover
- Use logging

**Actions by coordinator:**

```python
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
multicast GLOBAL_ABORT to all participants;
exit;
    }
record vote;
```
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
  write GLOBAL_COMMIT to local log;
  multicast GLOBAL_COMMIT to all participants;
} else {
  write GLOBAL_ABORT to local log;
  multicast GLOBAL_ABORT to all participants;
}
2PC: Participant recovery

actions by participant:

write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
send VOTE_COMMIT to coordinator;
wait for DECISION from coordinator;
if timeout {
    multicast DECISION_REQUEST to other participants;
    wait until DECISION is received; /* remain blocked */
    write DECISION to local log;
}
if DECISION == GLOBAL_COMMIT
    write GLOBAL_COMMIT to local log;
else if DECISION == GLOBAL_ABORT
    write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
send VOTE_ABORT to coordinator;
}
2PC: Participant recovery (cont’d)

Actions for handling decision requests: /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}
```

(b)

- Used to help other participants
3PC

- 2PC is very expensive
- Blocking after a failed node recovers to make a decision
- Add one more round: PREPARE-COMMIT
- Look at 3PC
Three-Phase Commit

(a) 

INIT → WAIT

Commit
Vote-request

Vote-abort
Global-abort

WAIT → ABORT

Vote-commit
Prepare-commit

WAIT → PRECOMMIT

Ready-commit
Global-commit

PRECOMMIT → COMMIT

(b) 

INIT → READY

Vote-request
Vote-abort

Vote-request
Vote-commit

READY → ABORT

Global-abort
ACK

READY → PRECOMMIT

Prepare-commit
Ready-commit

PRECOMMIT → COMMIT

Global-commit
ACK
What is a Byzantine Failure?

Three primary differences from Fail-Stop

1) Component can produce arbitrary output
   • Fail-stop: produces correct output or none

2) Cannot always detect output is faulty
   • Fail-stop: can always detect that component has stopped

3) Components may work together maliciously
   • No collusion across components
Agreement in Faulty Systems
Agreement in Faulty Systems

(a) Graph showing connections between nodes 1, 2, and 3 with arrows indicating communication paths.

Faulty process

(b) Log entries for nodes 1, 2, and 3:
1. Got(1, 2, x)
2. Got(1, 2, y)
3. Got(1, 2, 3)

(c) Log entries for faulty node 3 and other nodes:
1. Got(1, 2, y) (a, b, c)
2. Got(1, 2, x) (d, e, f)
Agreement in Faulty Systems

1. Got(1, 2, x, 4)
2. Got(1, 2, y, 4)
3. Got(1, 2, 3, 4)
4. Got(1, 2, z, 4)

1. Got (1, 2, y, 4)
   \[ (1, 2, y, 4) \]
2. Got (1, 2, x, 4)
   \[ (1, 2, x, 4) \]
3. Got (a, b, c, d)
   \[ (a, b, c, d) \]
4. Got (e, f, g, h)
   \[ (e, f, g, h) \]
5. Got (1, 2, z, 4)
   \[ (1, 2, z, 4) \]
6. Got (i, j, k, l)
   \[ (i, j, k, l) \]
General Impossibility Result

• No solution with fewer than $3m+1$ generals can cope with $m$ traitors
Recovery

- Recovery from failure
- Backward recovery: go back to a correct state
  - checkpointing; logging
- Forward recovery: make current or future state correct
  - plan for errors
Stable Storage

Disk **data errors** -> write during a crash, spontaneous bit error not handled by RAID errors detected by ECC upon read

Operations for stable storage using 2 identical disks

(spontaneous error: 1 drive only)
Stable Storage: writes/reads

Stable writes
Write 1 disk, then read it back, check ECC, do N times until it works; get a spare disk if not
Write 2, ....

Stable reads
read disk 1, if ECC fails, try N times, else read disk 2, ...
since can’t have 2 disk errors, will succeed
Stable Storage: crash recovery

- Spontaneous bit errors (in 1 drive) are no problem (stable read)
Checkpointing

Challenge: distributed recover to most recent consistent distributed snapshot
Independent Checkpointing

- The domino effect
Coordinated Checkpoints

• Avoid cascading rollbacks
• Use 2PC - how?

• Checkpoints are large
• Instead save messages and replay
Assumptions

• Deterministic state
• Given a prior state and a log of messages
• Final state will be the same after replay