Byzantine Fault Tolerance

CS 425: Distributed Systems
Fall 2012
Lecture 26
November 29, 2012
Presented By: Imranul Hoque
Reading List


Problem

• Computer systems provide crucial services
• Computer systems fail
  – Crash-stop failure
  – Crash-recovery failure
  – Byzantine failure
• Example: natural disaster, malicious attack, hardware failure, software bug, etc.
• Need highly available service

Replicate to increase availability
Byzantine Generals Problem

- All loyal generals decide upon the same plan
- A small number of traitors can’t cause the loyal generals to adopt a bad plan

Solvable if more than two-third of the generals are loyal
Practical Byzantine Fault Tolerance

- Before PBFT: BFT was considered too impractical in practice
- Practical replication algorithm
  - Weak assumption (BFT, asynchronous)
  - Good performance
- Implementation
  - BFT: A generic replication toolkit
  - BFS: A replicated file system
- Performance evaluation

Byzantine Fault Tolerance in Asynchronous Environment
Challenges

Request A

Client

Request B

Client
Challenges

1: Request A
2: Request B
How to assign sequence number to requests?
Primary Backup Mechanism

What if the primary is faulty?
Agreeing on sequence number
Agreeing on changing the primary (view change)

1: Request A
2: Request B
Agreement

- Certificate: set of messages from a quorum
- Algorithm steps are justified by certificates
Algorithm Components

- Normal case operation
- View changes
- Garbage collection
- State transfer
- Recovery

All have to be designed to work together
Normal Case Operation

• Three phase algorithm:
  – PRE-PREPARE picks order of requests
  – PREPARE ensures order within views
  – COMMIT ensures order across views

• Replicas remember messages in log

• Messages are authenticated
  – \{.\}_{ok} denotes a message sent by k

Quadra&c message exchange
Pre-prepare Phase

Request: m

{PRE-PREPARE, v, n, m}_{00}

Primary: Replica 0

Replica 1

Replica 2

Replica 3

Fail
Prepare Phase

Request: m

Primary: Replica 0

Replica 1

Replica 2

Replica 3

Failed

Accepted PRE-PREPARE
Prepare Phase

Request: m

Primary: Replica 0

Replica 1

Replica 2

Replica 3

Prepared PRE-PREPARE

{PREPARE, v, n, D(m), 1}_{\sigma_1}

Accepted PRE-PREPARE

Fail
Prepare Phase

- Request: m
- Primary: Replica 0
- Replica 1
- Replica 2
- Replica 3
- Fail
- Accepted PRE-PREPARE

Collect PRE-PREPARE + 2f matching PREPARE
Commit Phase

Request: m

PRE-PREPARE

PREPARE

Primary: Replica 0

Replica 1

Replica 2

Replica 3

Fail

{COMMIT, v, n, D(m)}_{o2}
Commit Phase (2)

Request: m

Collect 2f+1 matching COMMIT: execute and reply

PRE-PREPARE

PREPARE

COMMIT

Primary: Replica 0

Replica 1

Replica 2

Replica 3 Fail
View Change

• Provide liveness when primary fails
  – Timeouts trigger view changes
  – Select new primary (= view number mod 3f+1)

• Brief protocol
  – Replicas send VIEW-CHANGE message along with the requests they prepared so far
  – New primary collects 2f+1 VIEW-CHANGE messages
  – Constructs information about committed requests in previous views
View Change Safety

• **Goal:** No two different committed request with same sequence number across views

Quorum for Committed Certificate \((m, v, n)\)

View Change Quorum

At least one correct replica has Prepared Certificate \((m, v, n)\)
Recovery

• Corrective measure for faulty replicas
  – Proactive and frequent recovery
  – All replicas can fail if at most f fail in a window

• System administrator performs recovery, or

• Automatic recovery from network attacks
  – Secure co-processor
  – Read-only memory
  – Watchdog timer

Clients will not get reply if more than f replicas are recovering
Sketch of Recovery Protocol

• Save state
• Reboot with correct code and restore state
  – Replica has correct code without losing state
• Change keys for incoming messages
  – Prevent attacker from impersonating others
• Send recovery request $r$
  – Others change incoming keys when $r$ execute
• Check state and fetch out-of-date or corrupt items
  – Replica has correct up-to-date state
Optimizations

- Replying with digest
- Request batching
- Optimistic execution
Performance

- Andrew benchmark
  - Andrew100 and Andrew500
- 4 machines: 600 MHz, Pentium III
- 3 Systems
  - BFS: based on BFT
  - NO-REP: BFS without replication
  - NFS: NFS-V2 implementation in Linux

No experiment with faulty replicas
Scalability issue: only 4 & 7 replicas
Benchmark Results

Fig. 15. Andrew100 and Andrew500: elapsed time in seconds.

Without view change and faulty replica!
Questions?