View Change Protocols and Reconfiguration

Material by: Kyle Jamieson
(Used with permission)
Primary-backup replication

- Nominate one replica *primary*
  - Clients send all requests to *primary*
  - Primary orders clients’ requests

From two to many

- Primary-Backup case study
- State Machine Replication with *many* replicas
  - Survive *more* failures
Introduction to Viewstamped Replication

- **State Machine Replication** for any number of replicas

- **Replica group**: Group of $2f + 1$ replicas
  - Protocol can tolerate $f$ replica crashes

  **Viewstamped Replication Assumptions:**

  1. Handles **crash failures** only
     - Replicas fail only by **completely stopping**

  2. **Unreliable network**: Messages might be lost, duplicated, delayed, or delivered out-of-order

Replica state

1. **configuration**: identities of all $2f + 1$ replicas

2. In-memory **log** with clients’ requests in assigned order

   $\langle \text{op1, args1} \rangle \langle \text{op2, args2} \rangle \langle \text{op3, args3} \rangle \langle \text{op4, args4} \rangle$
1. Primary adds request to end of its log
2. Replicas add requests to their logs in primary’s log order
3. Primary \textit{waits for }$f$\textit{PrepareOKs }$\rightarrow$\textit{request is committed}

Normal operation: Key points

- Protocol guarantees \textbf{state machine replication}
- On \texttt{execute}, primary knows request in $f + 1 = 2$ nodes’ logs
  – Even if $f = 1$ then \texttt{crash}, $\geq 1$ retains request in log
Where's the commit message? \((f = 1)\)

- Previous Request's commit *piggybacked* on current *Prepare*

- No client Request after a timeout period?
  - Primary sends *Commit* message to all backups

The need for a view change

- So far: *Works* for *f* failed backup replicas

- But what if the *f* failures include a *failed primary*?
  - All clients' requests go to the *failed primary*
  - System *halts* despite *merely* *f* failures
Today

1. Primary-backup replication

2. View changes
   - With Viewstamped Replication
   - Using a View Server
   - Failure detection

Views

• Let different replicas assume role of primary over time

• System moves through a sequence of views
  - View = (view number, primary id, backup id, ...)

View #1

View #2

View #3
View change protocol

• Backup replicas monitor primary

• If primary seems faulty (no Prepare/Commit):
  – Backups execute the view change protocol to select new primary
  • View changes execute automatically, rapidly

• Need to keep clients and replicas in sync: same local state of the current view
  • Same local state at clients
  • Same local state at replicas

Making the view change correct

• View changes happen locally at each replica

• Old primary executes requests in the old view, new primary executes requests in the new view

• Want to ensure state machine replication

• So correctness condition: Executed requests
  1. Survive in the new view
  2. Retain the same order in the new view
Replica state (for view change)

1. **configuration**: sorted identities of all $2f + 1$ replicas
2. In-memory log with clients’ requests in assigned order
3. **view-number**: identifies primary in configuration list
4. **status**: normal or in a view-change

View change protocol

1. B notices A has failed, sends **Start-View-Change**
2. C replies **Do-View-Change** to new primary, with its log
3. B waits for $f$ replies, then sends **Start-View**
4. On receipt of Start-View, C replays log, accepts new ops
View change protocol: Correctness \((f = 1)\)

- Old primary \(A\) must have received one or two \texttt{PrepareOK} replies for that request (\emph{why}?)

- Request is in \(B\)'s or \(C\)'s \texttt{log} (or both): so it \texttt{will survive} into new view

Principle: Quorums \((f = 1)\)

- Any \textbf{group of} \(f + 1\) \textbf{replicas} is called a \textbf{quorum}

- \textbf{Quorum intersection property}: Two quorums in \(2f + 1\) replicas must \textbf{intersect} at \textbf{at least one replica}
Applying the quorum principle

Normal Operation:

• Quorum that processes one request: $Q_1$
  – ...and 2\textsuperscript{nd} request: $Q_2$

• $Q_1 \cap Q_2$ has at least one replica →
  – Second request reads first request’s effects

Applying the quorum principle

View Change:

• Quorum processes previous (committed) request: $Q_1$
  – ...and that processes Start-View-Change: $Q_2$

• $Q_1 \cap Q_2$ has at least one replica →
  – View Change contains committed request
Today

1. Primary-backup replication

2. View changes
   - With Viewstamped Replication
   - Using a View Server
   - Failure detection

Would centralization simplify design?

• A single View Server could decide who is primary
  – Clients and servers depend on view server
    • Don't decide on their own (might not agree)

• Goal in designing the VS:
  – Only want one primary at a time for correct state machine replication
View Server protocol operation

• For now, assume VS never fails

• Each replica now periodically pings the VS
  – VS declares replica dead if missed N pings in a row
  – Considers replica alive after a single ping received

• Problem: Replica can be alive but because of network connectivity, be declared “dead”

View Server: Split Brain

23

24
One possibility: $S_2$ in old view

Also possible: $S_2$ in new view
Split Brain and view changes

Take-away points:

- Split Brain problem can be avoided both:
  - In a decentralized design (VR)
  - With centralized control (VS)

- But protocol must be designed carefully so that replica state does not diverge

Today

1. Primary-backup replication

2. View changes
   - With Viewstamped Replication
   - Using a View Server
   - Failure detection
Both crashes and network failures are frequent: the “common case”

Q: How does one replica estimate whether another has crashed, or is still alive?

A: Failure detection algorithm
   - So far, we’ve seen Viewstamped Replication e.g.:
     • Replicas listen for Prepare or Commit messages from the Primary
     • Declare primary failed when hear none for some period of time

Failure detection: Goals

• Completeness: Each failure is detected
• Accuracy: There is no mistaken detection
• Speed: Time to first detection of a failure
• Scale (if significant in system context):
  • Equal processing load on each node
  • Equal network message load
Centralized versus Gossip

- **Centralized**
  - C thinks X is dead

- **Gossip**
  - Overcomes failure

"X is alive."