Proactive Recovery in a Byzantine-Fault-Tolerant System

Paper by Miguel Castro and Barbara Liskov
Presentation by Paul Kuliniewicz
Problem

- Ordinary Byzantine-fault-tolerant systems can only handle $f$ faults during the lifetime of the system.
- Once $f + 1$ faults occur, the system fails.
- If we can recover faulty replicas, the system can handle more than $f$ faults (as long as no more than $f$ occur in any given window).
- But how do we recover in a Byzantine environment?
Approach

- **Proactive recovery** – We can’t reliably detect faults, so periodically recover replicas.

- **Fresh messages** – Old messages could’ve come from a faulty replica, so ignore messages that are too old.

- **Efficient state transfer** – Recovered replicas need to bring their state up to date quickly.
Faulty replicas can:

- behave arbitrarily
- coordinate with each other
- delay messages
- inject messages
- delay correct replicas (but not indefinitely)
Model: Network

- Clients and replicas communicate by sending messages.

- Messages:
  - are asynchronous
  - can be dropped
  - can be delayed
  - can be duplicated
  - can arrive out of order
Model: Replicas

- Secure Cryptographic Coprocessor
  - Signs and decrypts using inaccessible private key
  - True random number generator
  - Reliable counter
- Read-Only Memory
  - Stores public keys and program code
- Watchdog Timer
  - Periodically triggers recovery monitor
  - Tamperproof without physical access
The system is a distributed state machine.
All operations are deterministic.
All correct replicas must agree on the order of operations.
Definitions

- $\mathcal{R}$ is the set of replicas, numbered $0$ through $|\mathcal{R}| - 1$.

- One view $v$ is active at a time. One replica is the primary $p$ of the view, and the others are backups.
  
  - $p = v \mod |\mathcal{R}|$

- A certificate is a collection of messages from different replicas that certify that some statement is correct.
Keys

- Each system has a fixed public and private key.

- \( k_{i,j} \) – session key used to send from replica \( i \) to replica \( j \).

  - \( k_{i,j} \neq k_{j,i} \)

- \( k_{c,i} \) – session key used to send from client \( c \) to replica \( i \).

  - \( k_{c,i} = k_{i,c} \)

- Session keys can (and will!) change over time.
Crypto Notation

For some message $m$:

- $\langle m \rangle_{\mu_{i,j}} - m$ with MAC using $k_{i,j}$.
- $\langle m \rangle_{\alpha_i} - (\langle m \rangle_{\mu_{i,1}}, \ldots, \langle m \rangle_{\mu_{i,|R|-1}})$.
- $\langle m \rangle_{\sigma_i} - m$ signed with $i$’s private key.
- $\langle m \rangle_{\epsilon_j} - m$ encrypted with $j$’s public key.
High-Level Overview

- Same basic idea as in "Practical Byzantine Fault Tolerance," but:
  - Replicas are automatically recovered at regular intervals.
  - (Almost) all authentication done using MACs.
  - View changes are more complicated.
Key Exchange

- Replica $i$ generates new inbound session keys and multicasts:
  \[
  \langle \text{NEW-KEY}, i, \ldots, \{k_{j,i}\}_{\epsilon_j, \ldots, t}\rangle_{\sigma_i}.
  \]

- Secure counter $t$ prevents replay attacks.

- New keys generated when replica is recovered, or periodically otherwise.

- After sending new keys, replicas throw away messages in incomplete certificates in log authenticated with old keys.
  - Guarantees all messages in a certificate are equally fresh.
Key Exchange (2)

- If $j$ can’t authenticate a message it receives from $i$, $j$ resends its last new-key message to $i$.

- Clients use similar procedure to distribute their session keys.
Processing Requests

- Same basic three-phase procedure as before:
  - Client $c$ sends $\langle \text{REQUEST}, o, t, c \rangle_{\alpha_c}$.
  - Primary chooses $n$ and multicasts $\langle \langle \text{PRE-PREPARE}, v, n, d \rangle_{\alpha_p}, m \rangle$.
  - Replica $i$ multicasts $\langle \text{PREPARE}, v, n, d, i \rangle_{\alpha_i}$.
  - Replica $i$ multicasts $\langle \text{COMMIT}, v, n, d, i \rangle_{\alpha_i}$.
  - Replica $i$ executes and sends $\langle \text{REPLY}, v, t, c, i, r \rangle_{\mu_{i,c}}$ to client.
More Déjà Vu:

- Every $K$ requests, replica $i$ multicasts $\langle \text{CHECKPOINT}, n, d, i \rangle_{\alpha_i}$.
- Upon receiving $2f + 1$ valid checkpoint messages for $n$, the checkpoint is stable; delete log entries $\leq n$.
- Adjust water marks: $h = n$, $H = h + L$. 

Checkpointing
View Changes

- Replicas change the current view when they suspect the primary has failed.
- All correct replicas agree on committed sequence numbers across view changes.
- Views are maintained long enough to make some progress.
- Ideas the same as before, but implementation is complicated by the limited lifetime of certificates.
Recovery

- Watchdog timer triggers recovery monitor.
- Log and system state are saved to disk.
- Replica is rebooted from code stored in ROM.
- Replica loads log and system state from disk.
  - But there’s no guarantee *these* are correct!
Recovering from Recovery

- Replica must process requests as soon as its code is loaded.
  - Otherwise it’s essentially a failed replica.
  - If $f$ are currently failed, we can’t be number $f + 1$!

- Yet need to verify (or correct!) log and system state before executing any operations.

- Worse, the attacker could have the stored session keys and can forge messages to and from the replica!
Step 1: Regenerating Keys

- Must assume that all session keys have been compromised.
- Throw away all client session keys.
- Generate new session keys for the other replicas and distribute them.
Step 2: Finding $H_M$

- Need an upper bound $H_M$ on high-water mark to discard bogus log entries.
- $i$ multicasts $\langle \text{QUERY-STABLE}, i, r \rangle_{\alpha_i}$.  
  - $r$ is a random nonce.
- Other replicas $j$ reply with $\langle \text{REPLY-STABLE}, c, p, i, r \rangle_{\mu_{j,i}}$.  
  - $c =$ sequence number of last checkpoint. 
  - $p =$ sequence number of last prepared request.
- $i$ keeps doing this, keeping the smallest $c$ and maximum $p$ from each replica.
Step 2: Finding $H_M$

(2)

- $i$ chooses $c_M$ to be the $c$ value from replica $j$ such that:
  - $2f$ other replicas’ $c$ values are $\leq c_M$, and
  - $f$ other replicas’ $p$ values are $\geq c_M$.

- $i$ sets $H_M = c_M + L$.

- $c_M$ must be greater than any stable checkpoint.

- $c_M$ must be close to a correct replica’s checkpoint, to prevent faulty replicas from delaying recovery.
Step 3: Recovery Request

- $i$ issues request
  \[\langle \text{REQUEST}, \langle \text{RECOVERY}, H_M \rangle, t, i \rangle_{\sigma_i};\] gets assigned sequence number $n_R$.

- Other replicas issue new session keys when executing the request.
  - $H_R = \lfloor n_R/K \rfloor \cdot K + L$ bounds sequence number of possibly-forged messages from $i$.

- $i$ waits for $2f + 1$ replies (not just $f + 1$).

- $i$ computes recovery point $H = \max(H_M, H_R)$ and a valid view.
Step 4: Fetch State

- $i$ will be *recovered* once checkpoint $H$ is stable.
  - Guarantees that that state is held by $f + 1$ correct replicas.

- Other replicas know this too.
  - Changing session keys prevents bogus messages from $i$ with sequence numbers higher than $H$.  


State Transfer

- State transfer is needed when a replica needs to be brought up to date.
- Must be fast and efficient, since it’s needed each time a replica undergoes recovery.
- Must also guarantee replicas receive correct state information.
Representing State

- State is a contiguous memory range divided into *pages*.
- Trees represent state at each checkpoint:
  - Leaf nodes are individual pages.
  - Internal nodes store meta-data for children.
- Data at each node includes:
  - $lm$ – Sequence number of last checkpoint where node (or children) was modified.
  - $d$ – Digest of the node’s data.
  - $p$ – Data contained within the page (leaf nodes only).
State Tree
What’s in a Digest?

- Leaf nodes:
  - \( d = \text{MD5}(x, lm, p) \)

- Internal nodes:
  - \( d = \text{MD5}(x, lm, d_1 + \ldots + d_n) \)
  - \( d_1, \ldots, d_n \) are digests of the child nodes
  - Modular sum is used

- Can verify digests of child nodes if the digest of the parent is known.
Trees are Helpful

- Replicas can traverse trees when transferring state, only transferring nodes that have changed.

- Copy-on-write can be used to reduce the amount of data that must be stored for checkpoints.
Fetching State

- $i$ multicasts $\alpha_i \langle \text{FETCH}, l, x, lc, c, k, i \rangle$ to get information at index $x$ of level $l$.

- $lc$ is last checkpoint number $i$ knows about.

- If $c \neq -1$, $i$ wants replica $k$ to send it the value of the node as of checkpoint $c$.

- $i$ only sets $c \neq -1$ if it already knows the digest of the node it’s requesting.
Saving Internal Nodes

- If $k$ has a checkpoint $c$, it replies with $\langle$META-DATA, $c$, $l$, $x$, $P$, $k$ $\rangle$.
  - $P = \{\langle x', lm, d \rangle | x'$ child of $x$ AND $lm > lc \}$
  - No MAC needed, since $i$ can verify the digests using the parent node’s digest.

- Replicas $\neq k$ reply only if they have a stable checkpoint greater than $lc$ or $c$.

- $i$ retries with different $k$ until $k$ sends a valid reply or $i$ gets $f + 1$ identical non-$k$ replies.
Fetching Leaf Nodes

- Works same as fetching internal nodes, except:
  - Metadata is for that node, not its (nonexistent) children.
  - $k$ replies with $\langle \text{DATA}, x, p \rangle$.

- This is efficient; the state itself only gets transferred once!

- $i$ keeps fetching more nodes until its state tree is up to date.
Fetching Example
Vulnerability Window

Vulnerability window is \( T_v = 2T_k + T_r \).

- \( T_k \) – maximum period between key refreshes
- \( T_r \) – maximum time needed for recovery

Little control over \( T_r \) during a DoS attack.

\( T_k \) strongly influenced by watchdog timeout period \( T_w \).

But tradeoff between security and performance.
Practical Considerations

- $T_w = 4 \cdot s \cdot R_n$ is suggested.
  - $R_n$ is average recovery time under normal load.
  - $s$ is a safety factor.

- Stagger replicas’ recovery cycles.

- Avoid monoculture of replica implementations.
  - Want probabilities of replica failures to be independent of each other.
The Real World

<table>
<thead>
<tr>
<th>system</th>
<th>Andrew100</th>
<th>Andrew500</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS-rec</td>
<td>443.5</td>
<td>2257.8</td>
</tr>
<tr>
<td>BFS</td>
<td>381.3</td>
<td>2202.9</td>
</tr>
<tr>
<td>NFS-std</td>
<td>332.0</td>
<td>1781.6</td>
</tr>
</tbody>
</table>

- Assumes 30 sec reboots and $T_k = 15$ sec.
- BFS-rec is 16% slower than BFS in Andrew100.
- BFS-rec is 2% slower than BFS in Andrew500.
Contributions

A distributed state machine that can tolerate any number of Byzantine faults as long as:

- No more than $f$ faults occur within any window, and
- Failed replicas can be recovered

Algorithm kept relatively efficient by avoiding expensive cryptographic operations whenever possible.