Depot: Cloud Storage with Minimal Trust

Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement,
Lorenzo Alvisi, Mike Dahlin, Michael Walfish

Presenter: Jimmy You
Outline

• Background
• Motivation
• How does Depot work?
• What properties does Depot provide?
• Evaluation
• Conclusion
Outline

• **Background**
• **Motivation**
• **How does Depot work?**
• **What properties does Depot provide?**
• **Evaluation**
• **Conclusion**
Cloud Storage

• Cloud Computing:
  • Software as a Service

• Cloud Storage Service:
  • Storage service backed by cloud
  • Typically Key-Value store
  • Typical API: GET() PUT()
Risks of Cloud Storage
Risks of Cloud Storage

• Storage Service Provider could be malicious
  • Not a focus in this paper

• Even with benign service, things could go wrong
  • Software bugs
  • Total failure
  • Network partition
  • …
Outline

• Background
• **Motivation**
• How does Depot work?
• What properties does Depot provide?
• Evaluation
• Conclusion
What can we trust?

• Ideality: only trust ourselves
  • Result: replication on all clients (no cloud at all)

• Trade-off: Availability v.s. Fault-tolerance
  • We want to provide as much as we can, while trust as less as we can.

• Depot’s approach: a radical fault-tolerance stance
Consistency level of current SSPs

- **Available**
  - Eventual: Amazon S3, Dynamo

- **Unavailable**
  - Linearizable: Azure, MegaStore

**Too weak:**
- Reads by a client can omit recent writes by the same client
- Dependent writes may be observed out-of-order

**Ideal:** GET return most recent PUT

- Cannot be enforced with:
  - High availability: CAP result
  - Minimal trust: Byzantine result

→ Difficult to program
Design Goals

• **Eliminates** trust for safety
  • A client will be safe as long as itself is correct.
  • Any subset of correct clients observe sensible semantics.

• **Minimizes** trust for liveness and availability
  • Eliminates trust for updates:
    • A client can always update authorized objects.
    • Correct, connected clients can always share updates.
  • Minimizes trust for read:
    • Can’t be eliminated due to a fundamental limit
Outline

• Background
• Motivation
• **How does Depot work?**
• What properties does Depot provide?
• Evaluation
• Conclusion
Depot: Design

• Three Key Ideas:
  1. Reduce misbehavior to concurrency
  2. Enforce Fork-Join-Causal (FJC) consistency
  3. Layer other storage properties over FJC
Key Idea 1: Reduce misbehavior to concurrency

- Faults can cause forks
- Forks taint correct nodes’ future updates
- Forks prevent eventual consistency
Key Idea 1:
Reduce misbehavior to concurrency

- Faulty node → Two (correct) virtual nodes
- Correct nodes can accept subsequent updates
- Correct nodes can evict faulty node
Faults v.s. Concurrency

• Faults $\rightarrow$ Concurrency
  • Allow correct nodes to converge
• Concurrency can introduce conflicts...
  • Already possible due to decentralized servers!
  • Handled by Apps (Apps for high availability allow concurrent writes)
• Depot exposes the conflicts to the application
  • GET operation returns set of most recent concurrent updates
Key Idea 2: Fork-Join-Causal Consistency

• Causal Consistency (CC)
  • If update $u_1$ by a node depends on an update $u_0$ by any node, then $u_0$ becomes observable before $u_1$ at any node.

• Fork-Join Causal (FJC) Consistency
  • If update $u_1$ by a correct node depends on an update $u_0$ by any node, then $u_0$ becomes observable before $u_1$ at any correct node.

• They are same in benign runs
• Add **metadata** to Puts
• Add local states to nodes
• Add **checks** on received metadata
The magic Metadata

\[ \text{dVV,}\{\text{key, H(value), logicalClock@nodeID, H(history)}\}\sigma \]

- dVV: dependency version vector.
- LogicalClock: advanced on every update at nodeID and also every successful update from peer (advanced to larger than peer's value).
- H(value): collision-resistant hash of the value rather than whole value.
- H(History): collision-resistant hash of most recent update by each node know to writer at that instant of issuing update.
Checks upon receiving

• Accept an update $u$ sent by $N$ only if:
  1. $u$ must be properly signed

  2. There is no omission
      • All updates in $u$’s history are also in local history

  3. History is not modified
      • $u$ is newer than any prior update by $N$
Summary of Design

• Protect safety
  • Local checks

• Protect liveness
  • Joining forks
  • Reduce failures to concurrency

• Fork-Join-Causal consistency
  • A novel consistency semantics
  • Suitable for environments with minimal trust
Outline

• Background
• Motivation
• How does Depot work?
• What properties does Depot provide?
• Evaluation
• Conclusion
Properties provided by Depot

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Safety/Liveness</th>
<th>Property</th>
<th>Correct nodes required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>Safety</td>
<td>Fork-Join Causal</td>
<td>Any subset</td>
</tr>
<tr>
<td></td>
<td>Safety</td>
<td>Bounded staleness</td>
<td>Any subset</td>
</tr>
<tr>
<td></td>
<td>Safety</td>
<td>Eventual consistency (s)</td>
<td>Any subset</td>
</tr>
<tr>
<td>Availability</td>
<td>Liveness</td>
<td>Eventual consistency (l)</td>
<td>Any subset</td>
</tr>
<tr>
<td></td>
<td>Liveness</td>
<td>Always write</td>
<td>Any subset</td>
</tr>
<tr>
<td></td>
<td>Liveness</td>
<td>Always exchange</td>
<td>Any subset</td>
</tr>
<tr>
<td></td>
<td>Liveness</td>
<td>Write propagation</td>
<td>Any subset</td>
</tr>
<tr>
<td></td>
<td>Liveness</td>
<td>Read availability / durability</td>
<td>A correct node has object</td>
</tr>
<tr>
<td>Integrity</td>
<td>Safety</td>
<td>Only auth. updates</td>
<td>Clients</td>
</tr>
<tr>
<td>Recoverability</td>
<td>Safety</td>
<td>Valid discard</td>
<td>Any subset</td>
</tr>
<tr>
<td>Eviction</td>
<td>Safety</td>
<td>Valid eviction</td>
<td>Any subset</td>
</tr>
<tr>
<td></td>
<td>Safety</td>
<td>Bounded forks</td>
<td>Any subset</td>
</tr>
</tbody>
</table>

Fig. 3. Summary of properties provided by Depot.
GET Availability & Durability

• Minimizes required number of correct nodes
  • Data can safely flow via any path
  • If any correct node has data, GET eventually succeeds.

• Make it likely that a correct node has data
  • SSP replicates to multiple servers
  • Additional replication to protect against total failure
Outline

• Background
• Motivation
• How does Depot work?
• What properties does Depot provide?
• Evaluation
• Conclusion
Evaluation Setup

• 12 nodes on local Emulab
  • 8 clients + 4 servers
  • 1Gbps Ethernet
• Each client issues 1 request/sec
  • Measure latency
  • Measure per-request cost
• Emulate traditional cloud storage
  • Servers implemented Depot without any checks
  • Clients don’t receive any metadata
### Evaluation Setup

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baseline</strong></td>
<td>Clients trust the server to handle their PUTs and GETs correctly. Clients neither maintain local state nor perform checks on returned data.</td>
</tr>
<tr>
<td><strong>B+Hash</strong></td>
<td>Clients attach SHA-256 hashes to the values that they PUT and verify these hashes on GETs.</td>
</tr>
<tr>
<td><strong>B+H+Sig</strong></td>
<td>Clients sign the values that they PUT and verify these signatures on GETs.</td>
</tr>
<tr>
<td><strong>B+H+S+Store</strong></td>
<td>The same checks as B+H+Sig, plus clients locally store the values that they PUT, for durability and availability despite server failures.</td>
</tr>
</tbody>
</table>

Baseline and its variants for comparison with depot
Evaluation Metrics

• Overhead
  • Performance overhead
  • Cost overhead

• Fault-tolerance
  • Effect of total server failure
Latency Overhead: Mean
Latency Overhead: Tail
Cost overhead: by $
Effect of Total Failure

![Graph showing the effect of total failure on staleness and GET latency over time. The graph compares Depot and B+H+S+Store scenarios.]
Outline

• Background
• Motivation
• How does Depot work?
• What properties does Depot provide?
• Evaluation
• Conclusion
Conclusion

• Depot: A radical stance for fault-tolerance
  • Any node could fail in any way
  • Eliminate trust for consistency, staleness, etc.
    • Any subset of correct connected clients get some guarantee
  • Minimize trust for GET availability
    • GET succeeds if any correct, reachable node has data
Thank You!

Q&A
Backup Slides
Sources of overhead in Depot

metadata = signature + partial VV + history hash

PUT

GET

SSP

metadata check = SHA256 check + RSA verify + history check

data check = SHA256 check