Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems

Wolf Honoré\textsuperscript{1} Jieung Kim\textsuperscript{1} Ji-Yong Shin\textsuperscript{2} Zhong Shao\textsuperscript{1}

\textsuperscript{1}Yale University
\textsuperscript{2}Northeastern University

OOPSLA 2021
Goal

Theorem \( KV_{correct} : correct \) KV.

Proof.

\[
\ldots
\]

Qed.
Network-Based Models Too Complex
Network-Based Models Too Complex

Challenges

Network errors

S1

prepare

time = 2
leader = true

{ "abc": "def",
  "foo": "bar"
}

commit

S2

time = 2
leader = false

ack

S3

time = 1
leader = false

{ "abc": "def",
  "foo": "bar"
}
Network-Based Models Too Complex

Challenges
- Network errors
- Protocol subtleties

S1
- time = 2
- leader = true
- prepare
- commit

S2
- time = 2
- leader = false
- ack

S3
- time = 1
- leader = false
- prepare
Network-Based Models Too Complex

Challenges
- Network errors
- Protocol subtleties
- Application bugs

Overview

ADO Model

DApps

End-to-End Verification

Conclusion

S1
- time = 2
- leader = true
- \{
  "abc": "def",
  "foo": "bar"
\}

S2
- time = 2
- leader = false
- commit

S3
- time = 1
- leader = false
- prepare

\{
  "abc": "def"
\}

\{
  "abc": "def",
  "foo": "bar"
\}

\{
  "abc": "def",
  "foo": "bar"
\}

S1 prepare time = 2 leader = true
S2 time = 2 leader = false
S3 time = 1 leader = false
State Machine Replication Too Abstract
Partial Failures

S1

{
  "abc": "def"
}

S2

{
  "abc": "def"
}

S3

{
  "abc": "def"
}
Partial Failures

S1
{
  "abc": "def",
  "foo": "bar"
}

S2
{
  "abc": "def"
}

S3
{
  "abc": "def"
}

Alice
"foo": "bar"
Partial Failures

S1
{
  "abc": "def",
  "foo": "bar"
}

S2
{
  "abc": "def"
}

S3
{
  "abc": "def"
}

Bob
Read
S2
S1
S3
Partial Failures
Partial Failures are Important

Partial failure is a central reality of distributed computing. [...] Being robust in the face of partial failure requires some expression at the interface level. (Jim Waldo. A Note on Distributed Computing. 1994)

- Unavoidable feature unique to distributed systems.
- Influence with all aspects of distributed protocols (e.g., leader election and reconfiguration).
- Can be used for performance optimizations.
  - TAPIR (SOSP ’15): Transactions with out-of-order commits.
  - Speculator (SOSP ’05): Speculative distributed file system.
A Sweet Spot?

State Machine Replication
- Hides partial failures.
- Abstracts protocol details.

Network-Based Models
- Shows partial failures.
- Abstracts protocol details.
- Blends protocol and application logic.
Contributions

- State Machine Replication
- ADO Model
- Network-Based Models

▶ ADO (atomic distributed object) model: a fault-aware and compositional abstraction.
Contributions

- ADO (atomic distributed object) model: a fault-aware and compositional abstraction.
- Advert: an end-to-end Coq verification framework.
- Several verified case studies, including a lock-free key-value store, and Two-Phase Commit with replicated resource managers.
### Contributions

**ADO Model**
- ADO (atomic distributed object) model: a fault-aware and compositional abstraction.
- Advert: an end-to-end Coq verification framework.
- Several verified case studies, including a lock-free key-value store, and Two-Phase Commit with replicated resource managers.
- Refinement with several Paxos variants, Chain Replication.
- Refinement with multi-Paxos C implementation.

**State Machine Replication**

**Network-Based Models**
Contributions

- **ADO (atomic distributed object) model**: a fault-aware and compositional abstraction.
- **Advert**: an end-to-end Coq verification framework.
- Several verified case studies, including a lock-free key-value store, and Two-Phase Commit with replicated resource managers.
- Refinement with several Paxos variants, Chain Replication.
- Refinement with multi-Paxos C implementation.
ADO State

ADO Legend

<table>
<thead>
<tr>
<th>Method</th>
<th>Timestamp</th>
<th>Persistent Log</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Paxos</td>
<td>1</td>
<td>&quot;abc&quot;: &quot;def&quot;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>&quot;foo&quot;: &quot;bar&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>

"xyz": "123" 4

S1
4
"abc": "def" 1
"foo": "bar" 2
S2
4
"abc": "def" 1
"foo": "bar" 2
S3
3
"cat": "dog" 3
"dot": "com" 3
S1
S2
S3
ADO State

ADO Legend

Method Timestamp

Persistent Log

Entry

Cache Tree

Entry

"abc": "def"
1

"foo": "bar"
2

"cat": "dog"
3

"xyz": "123"
4

"dot": "com"
3

S1

S2

S3
ADO Operations
ADO Operations

ADO

Multi-Paxos
Pull

ADO

```
"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
```

Multi-Paxos

```
"abc": "def"
1
"foo": "bar"
2
"cat": "dog"
3
"dot": "com"
3

S1
5

S2
4

S3
3

S1
S2
S3
```
Pull

ADO

Multi-Paxos
Pull

ADO

"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4

Multi-Paxos

S1
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4

S2
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4

S3
"abc": "def" 1
"foo": "bar" 2
"cat": "dog" 3
"dot": "com" 3
Pull

Get permission to update and select a starting point in the cache tree.
Invoke

ADO

ADO Model

DApps

End-to-End Verification

Conclusion

Multi-Paxos

Invoke a Method

Add a new entry to the cache tree.
Invoke

ADO

```
"abc": "def"  
1
"foo": "bar"  
2
"xyz": "123"  
4
```

Multi-Paxos

```
S1
5
"abc": "def"  
1
"foo": "bar"  
2
"xyz": "123"  
4
"bad": "cow"  
5
```

```
S2
5
"abc": "def"  
1
"foo": "bar"  
2
"xyz": "123"  
4
```

```
S3
5
"abc": "def"  
1
"foo": "bar"  
2
"cat": "dog"  
3
"dot": "com"  
3
"bad": "cow"  
5
"xyz": "123"  
4
```

```
S1  "bad": "cow"
```

```
S2
```

```
S3
```
Invoke

Invoking a Method
Add a new entry to the cache tree.
Invoke

Invoking a Method
Add a new entry to the cache tree.
Push

ADO

```
"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
```

Multi-Paxos

```
S1
"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
"bee": "gee"
5
"bad": "cow"
5
commit
```

S1

```
S2
"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
"bee": "gee"
5
```

S3

```
S2
"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
"bee": "gee"
5
```

```
S1
commit
```
Push

ADO

"abc": "def" 1
"foo": "bar" 2
"cat": "dog" 3
"dot": "com" 3
"xyz": "123" 4
"bee": "gee" 5
"bad": "cow" 5

Multi-Paxos

S1 5
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4
"bee": "gee" 5
"bad": "cow" 5

S2 5
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4
"bee": "gee" 5

S3 5
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4
"bee": "gee" 5

commit

S1 commit
S2
S3
Push

Move committed methods into the log and prune stale states from the tree.
Push

Move committed methods into the log and prune stale states from the tree.
Push

Move committed methods into the log and prune stale states from the tree.
Distributed Applications
Distributed Applications

```
ADO KV {
    shared kv : [string * int] := [];
    method set(k, v) { this.kv[hash(k)] := (v, len(v)); }
    method get(k) { return this.kv[hash(k)][0]; }
    method getmeta(k) { return this.kv[hash(k)][1]; }
}
```
Distributed Applications

ADO DVec[T] {
  shared data : [T] := [];
  method insert(idx, x) { this.data[idx] := x; }
  method get(idx) { return this.data[idx]; }
}

ADO DLock {
  shared owner : option N := None;
  method tryAcquire() { ... }
  method release() { ... }
}

DApp KVLock(lk: DLock, data: DVec[string], meta: DVec[int]) {
  proc set(k, v) {
    ... /* acquire, set data, set meta, release */
  }
  ... /* get, getmeta */
}
Distributed Applications

```r
DApp KVLock(lk: DLock, data: DVec[string], meta: DVec[int]) {
    proc set(k, v) {
        lk.pull();
    }
}
```
Distributed Applications

```plaintext
DApp KVLock(lk: DLock, data: DVec[string], meta: DVec[int]) {
    proc set(k, v) {
        while (lk.pull() == FAIL) {}
    }
}
```
Distributed Applications

```javascript
DApp KVLock(lk: DLock, data: DVec[string], meta: DVec[int]) {
    proc set(k, v) {
        while (lk.pull() == FAIL) {}
        ok := lk.tryAcquire();
    }
}
```
Distributed Applications

```plaintext
DApp KVLock(lk: DLock, data: DVec[string], meta: DVec[int]) {
    proc set(k, v) {
        while (lk.pull() == FAIL) {}
        ok := lk.tryAcquire();
        while (lk.push() == FAIL) {}
        if (!ok) { return; }
        /* ... */
    }
}
```
Method Calling Semantics

```plaintext
DApp KVLockAbort(lk: DLock, data: DVec[string], meta: DVec[int]) {
    proc set(k, v) {
        if (lk.pull() == FAIL) { return; }
        ok := lk.tryAcquire();
        if (lk.push() == FAIL) { return; }
        if (!ok) { return; }
        /* ... */
    }
}
```
Method Calling Semantics

```plaintext
DApp KVLockRetry(lk: DLock, data: DVec[string], meta: DVec[int]) {
    proc set(k, v) {
        for retry in 0..N {
            if (lk.pull() == FAIL) { continue; }
            ok := lk.tryAcquire();
            if (lk.push() == FAIL) { continue; }
            if (!ok) { continue; }
        }
        if (retry == N) { return; }
        /* ... */
    }
}
```
Method Calling Semantics

```
obj.m()! :=
  while (obj.pull() == FAIL) {}
obj.m();
while (obj.push() == FAIL) {}
```

DApp KVLock(lk: DLock, data: DVec[string], meta: DVec[int]) {
  proc set(k, v) {
    ok := lk.tryAcquire()!
    if (!ok) { return; }
    data.insert(hash(k), v)!
    meta.insert(hash(k), len(v))!
    lk.release()!
  }
}
```
Non-Standard Method Calls

DApp $TM(rm_1: RM, ..., rm_n: RM) \{$
  proc init() { // Must be called once when TM starts
    for rm in [this.rm_1, ..., this.rm_n] {
      while (rm.pull() == FAIL) {} // pull once up front
    }
  }

  proc collect_decisions(tx) {
    for rm in [this.rm_1, ..., this.rm_n] {
      rm.prepare(tx); // No pull needed
      for i in 0..MAX_TRY {
        res := rm.push(); // Only try up to MAX_TRY
        if (res != FAIL) { break; }
      }
      // Short-circuit on failure
      if (res == NO || res == FAIL) { tx.decision := ABORT; break; }
    }
  }
$
}
Non-Standard Method Calls

```java
DApp TM(rm_1: RM, ..., rm_n: RM) {
    proc init() { // Must be called once when TM starts
        for rm in [this.rm_1, ..., this.rm_n] {
            while (rm.pull() == FAIL) {} // pull once up front
        }
    }

    proc collect_decisions(tx) {
        for rm in [this.rm_1, ..., this.rm_n] {
            rm.prepare(tx); // No pull needed
            for i in 0..MAX_TRY {
                res := rm.push(); // Only try up to MAX_TRY
                if (res != FAIL) { break; }
            }
            // Short-circuit on failure
            if (res == NO || res == FAIL) { tx.decision := ABORT; break; }
        }
    }
}
```
Connection with Distributed Protocols
Refinement

- Pull
- Refine
- Prepare
- Invoke
- Local Update
- Push
- Commit
- Refine
Refinement
**Specification and Proof Effort**

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>KVLock</td>
<td>646</td>
</tr>
<tr>
<td>KVLockFree</td>
<td>359</td>
</tr>
<tr>
<td>2PC</td>
<td>559</td>
</tr>
<tr>
<td>Paxos-like</td>
<td>5K</td>
</tr>
<tr>
<td>Chain Replication</td>
<td>2K</td>
</tr>
<tr>
<td>Shared Libraries</td>
<td>11K</td>
</tr>
<tr>
<td>Single-Paxos</td>
<td>77</td>
</tr>
<tr>
<td>Multi-Paxos</td>
<td>87</td>
</tr>
<tr>
<td>Vertical Paxos</td>
<td>97</td>
</tr>
<tr>
<td>CASPaxos</td>
<td>78</td>
</tr>
</tbody>
</table>

Conclusion

▶ ADO Model: A novel, fault-aware, compositional distributed system abstraction.
▶ Advert: Coq framework for single- and multi-ADO reasoning.
▶ End-to-end guarantees with refinement.
▶ High-level behavior independent of underlying protocol.