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

Wolf Honoré[1] Jieung Kim[1] Ji-Yong Shin[2] Zhong Shao[1]

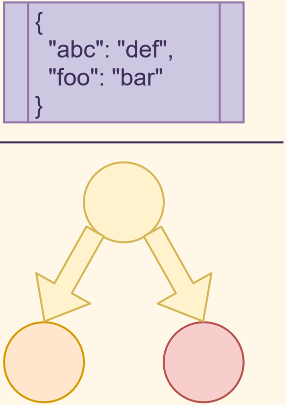
[1]Yale University

[2]Northeastern University

OOPSLA 2021

Goal

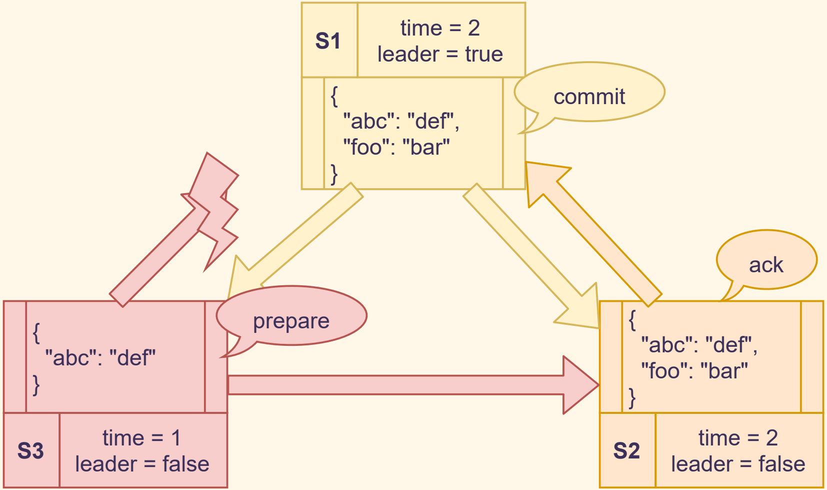
Application
Key-Value Store



Implementation
Multi-Paxos

```
Theorem KV_correct : correct KV.
Proof.
  ...
Qed.
```

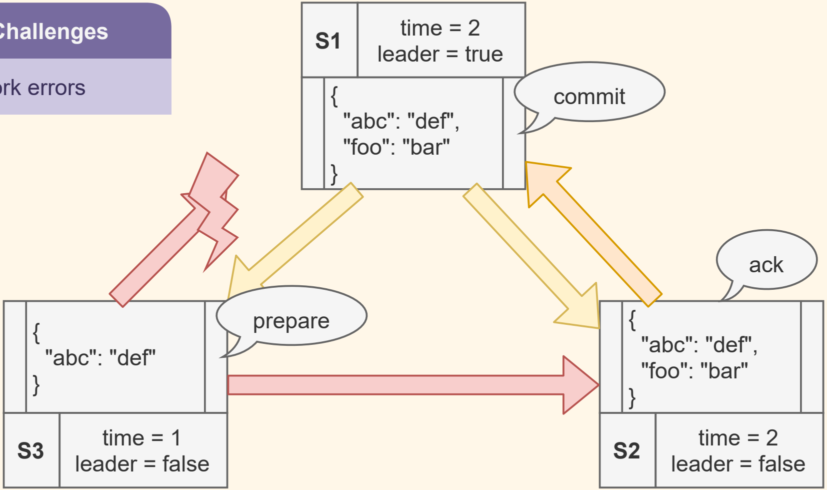
Network-Based Models Too Complex



Network-Based Models Too Complex

Challenges

Network errors

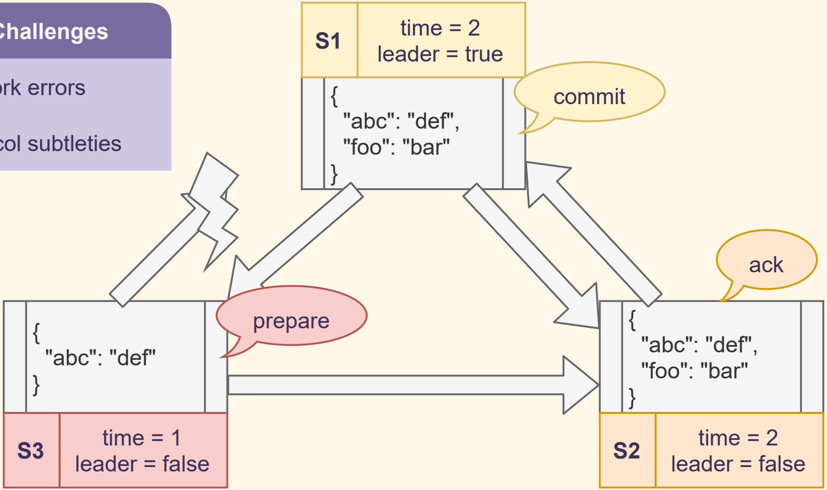


Network-Based Models Too Complex

Challenges

Network errors

Protocol subtleties



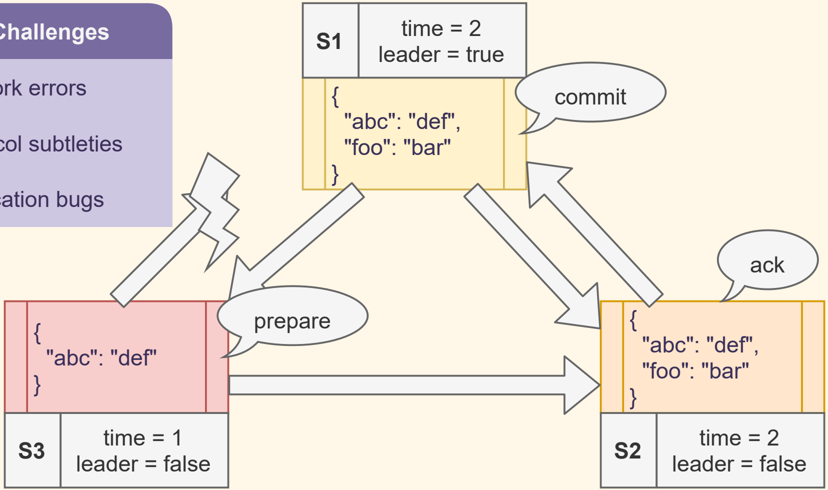
Network-Based Models Too Complex

Challenges

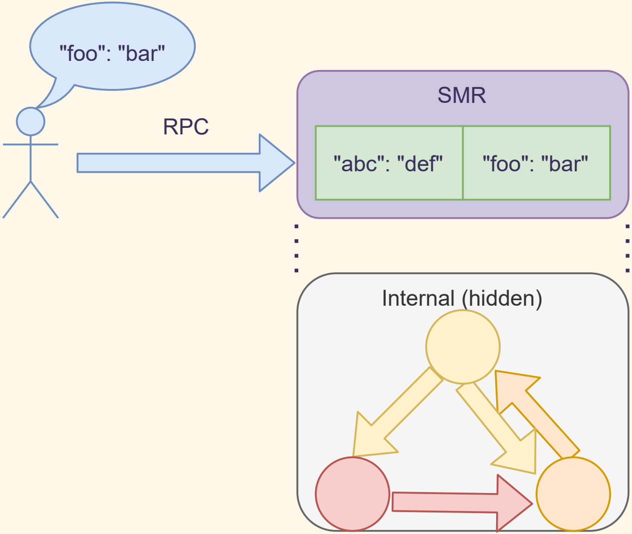
Network errors

Protocol subtleties

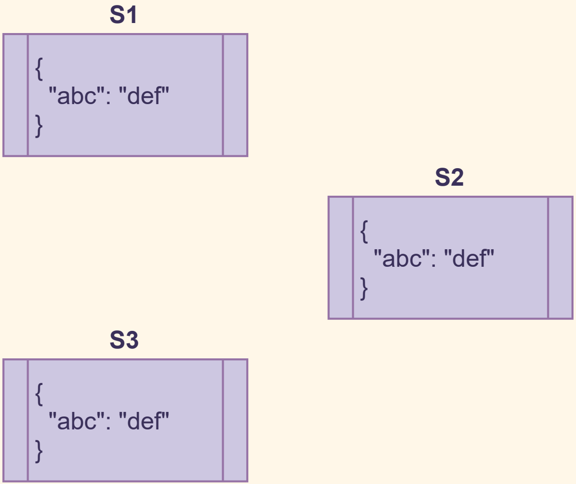
Application bugs



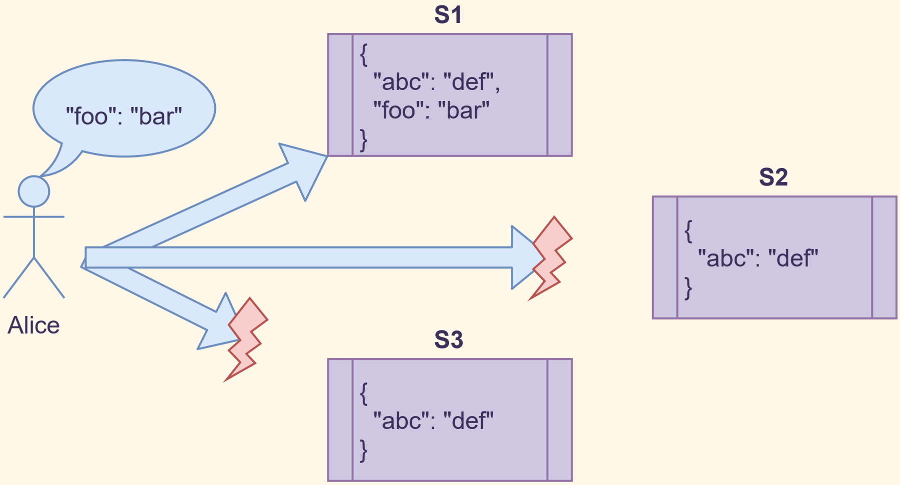
State Machine Replication Too Abstract



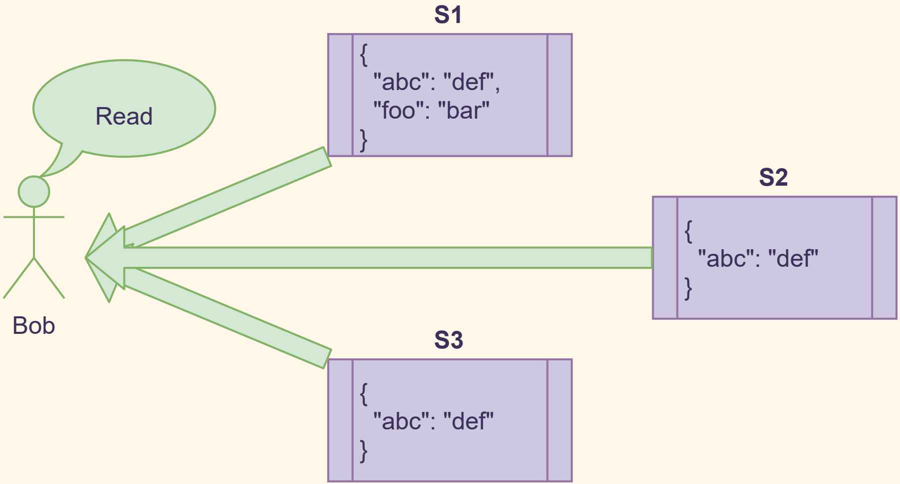
Partial Failures



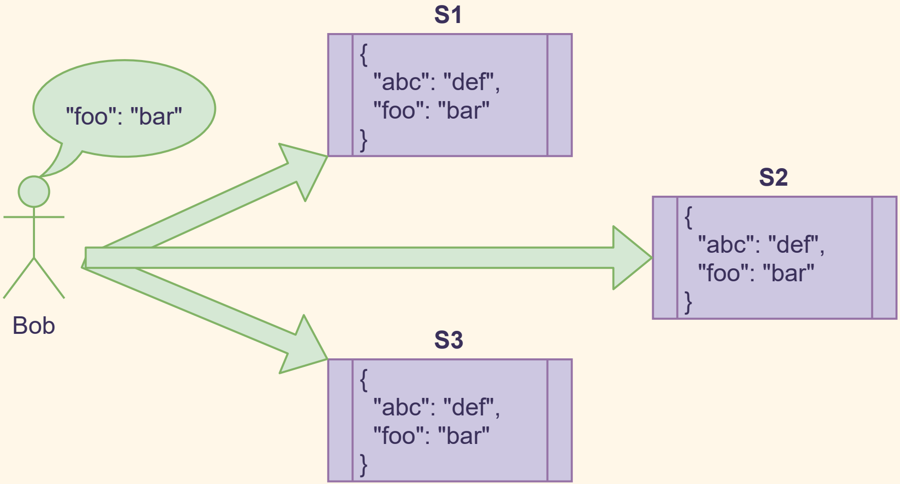
Partial Failures



Partial Failures



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.
?	✓ Shows partial failures. ✓ Abstracts protocol details.
Network-Based Models	✓ Shows partial failures. × 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

State Machine Replication

ADO Model

Network-Based Models

- 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

State Machine Replication

ADO Model

Network-Based Models

- 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.

Contributions

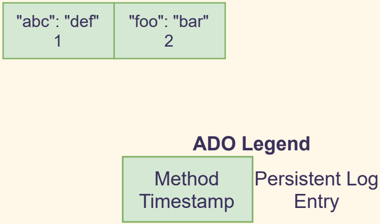
State Machine Replication

ADO Model

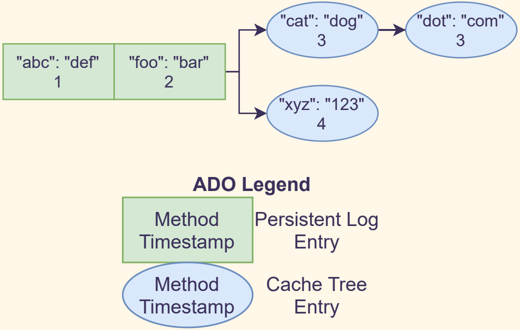
Network-Based Models

- 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.

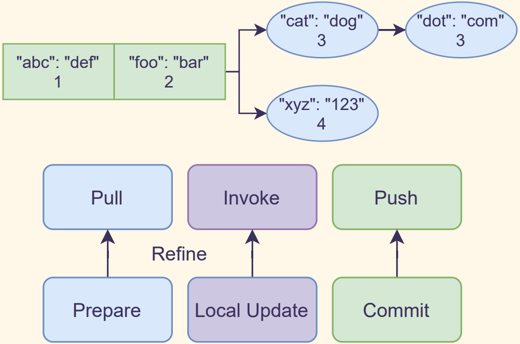
ADO State


"abc": "def"
1
"foo": "bar"
2
ADO Legend
Method
Timestamp
Persistent Log
Entry

ADO State

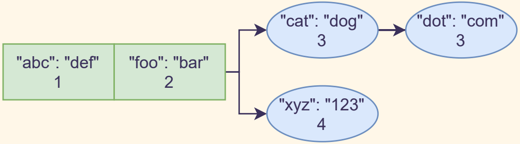

"abc": "def"
1
"foo": "bar"
2
"cat": "dog"
3
"dot": "com"
3
"xyz": "123"
4
ADO Legend
Method
Timestamp
Persistent Log Entry
Method
Timestamp
Cache Tree Entry

ADO Operations

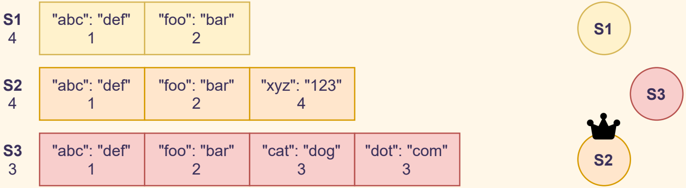


ADO Operations

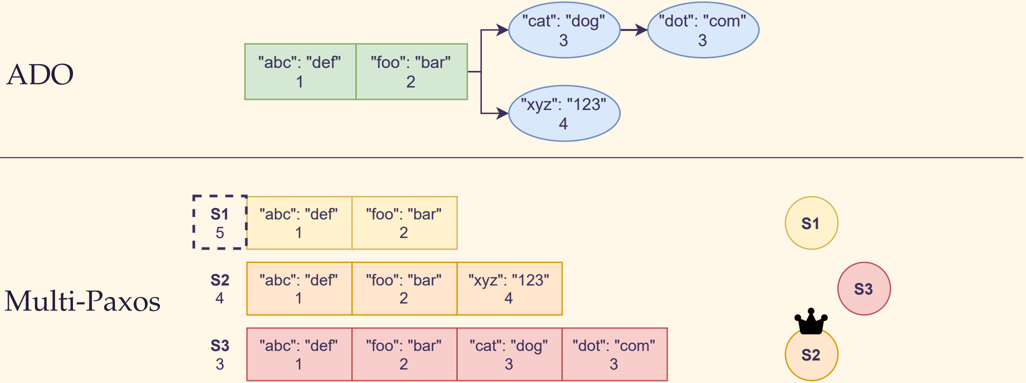
ADO



Multi-Paxos

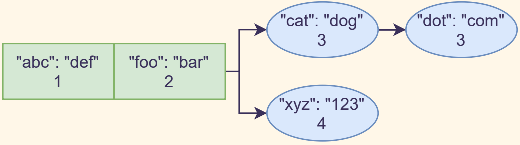


Pull

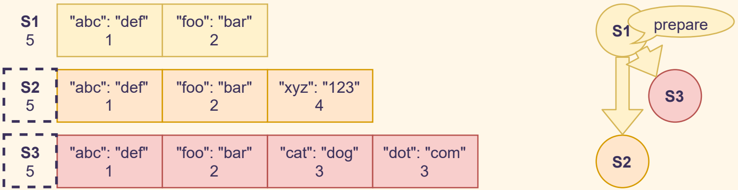

ADO
"abc": "def"
1
"foo": "bar"
2
"cat": "dog"
3
"dot": "com"
3
"xyz": "123"
4
Multi-Paxos
S1
5
"abc": "def"
1
"foo": "bar"
2
S2
4
"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
S3
3
"abc": "def"
1
"foo": "bar"
2
"cat": "dog"
3
"dot": "com"
3
S1
S3
S2

Pull

ADO

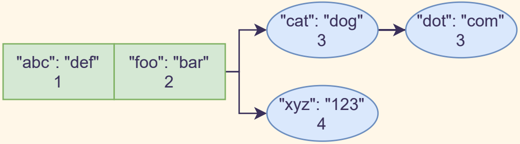


Multi-Paxos

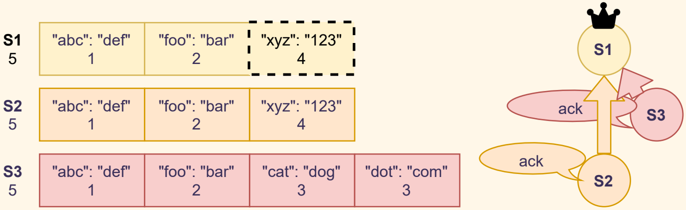


Pull

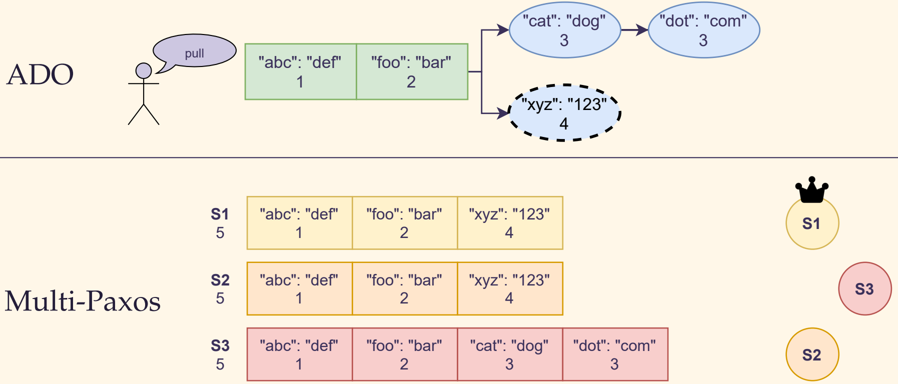
ADO



Multi-Paxos



Pull

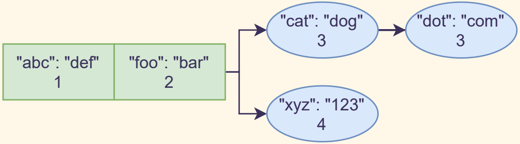


Pull

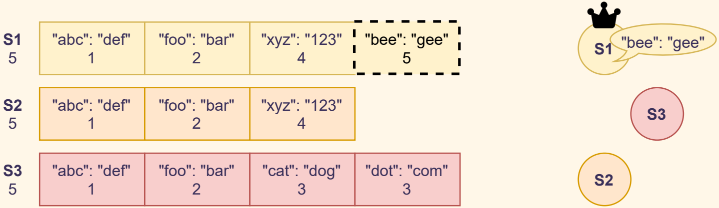
Get permission to update and select a starting point in the cache tree.

Invoke

ADO

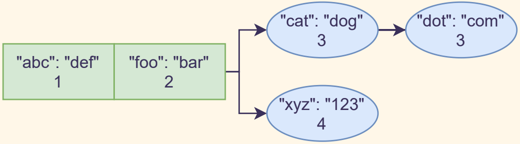


Multi-Paxos

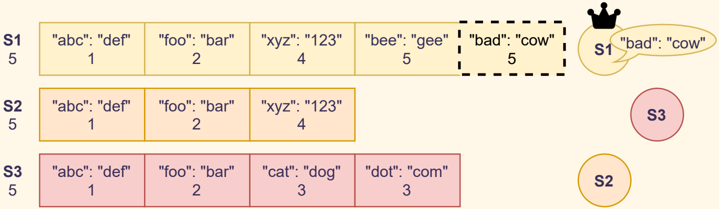


Invoke

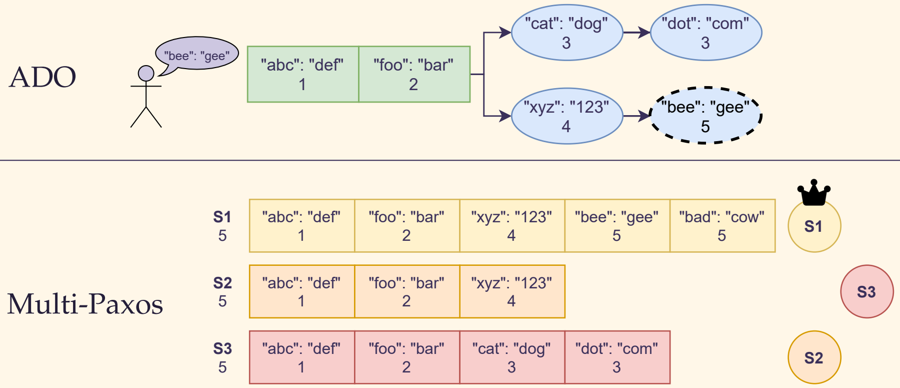
ADO



Multi-Paxos



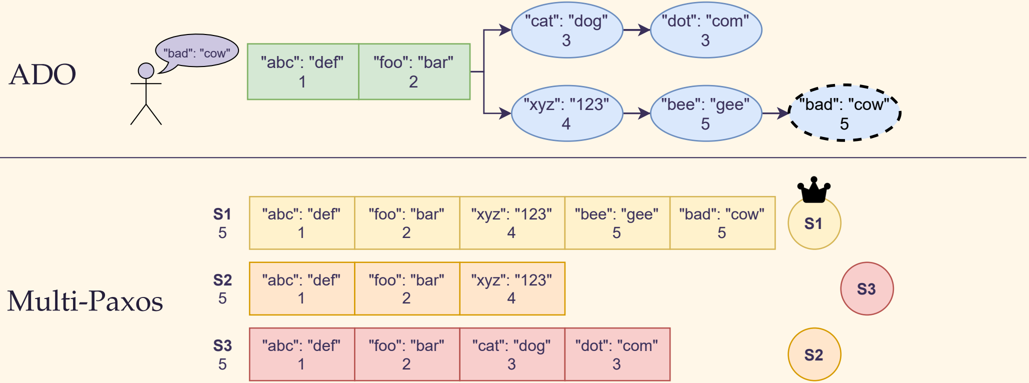
Invoke



Invoking a Method

Add a new entry to the cache tree.

Invoke

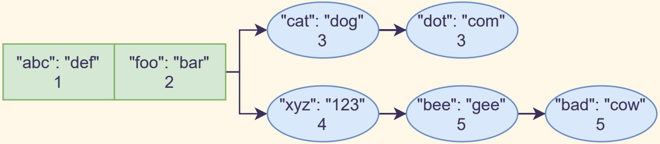


Invoking a Method

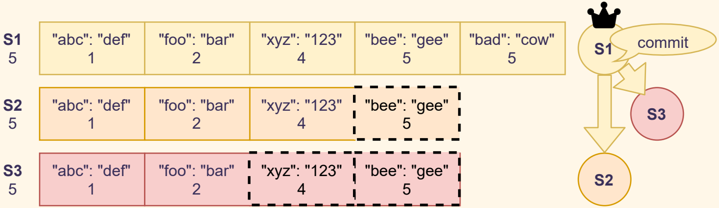
Add a new entry to the cache tree.

Push

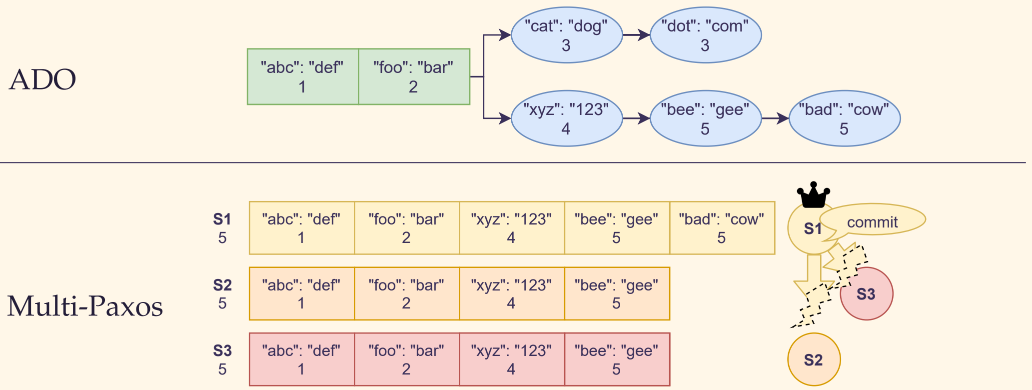
ADO



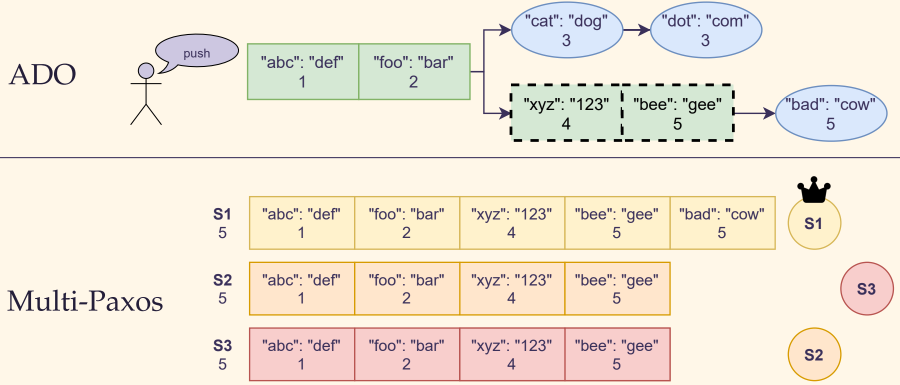
Multi-Paxos



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
S1
commit
S3
S2

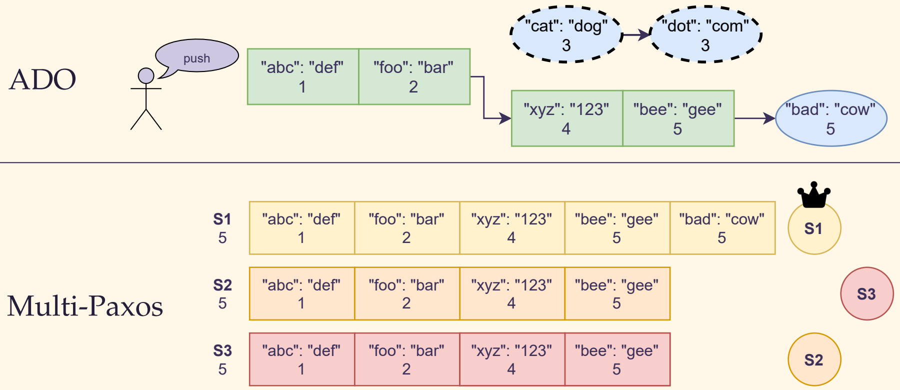
Push



Push

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

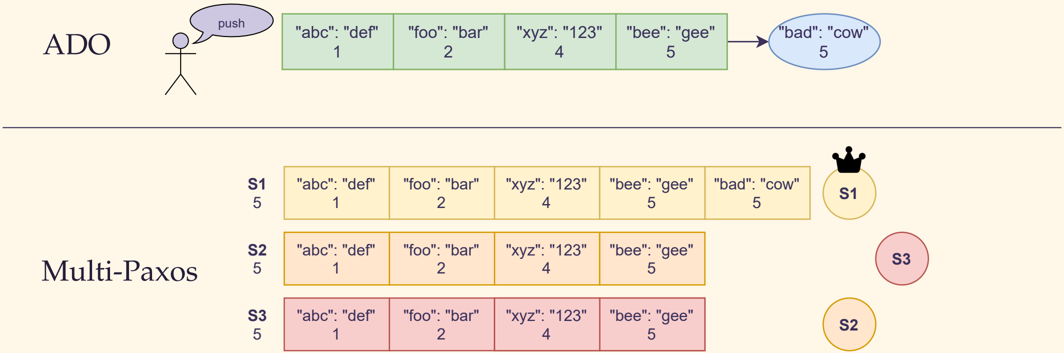
Push



Push

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

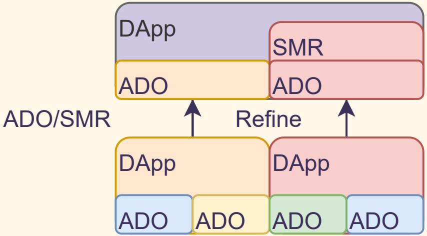
Push



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

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




  }
}
```

Distributed Applications

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




  }
}
```

Distributed Applications

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




  }
}
```

Distributed Applications

```
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

```
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

```
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()!;
  }
}
```

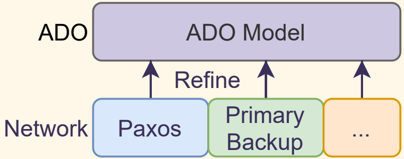
Non-Standard Method Calls

```
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; }
    }
  }
}
```

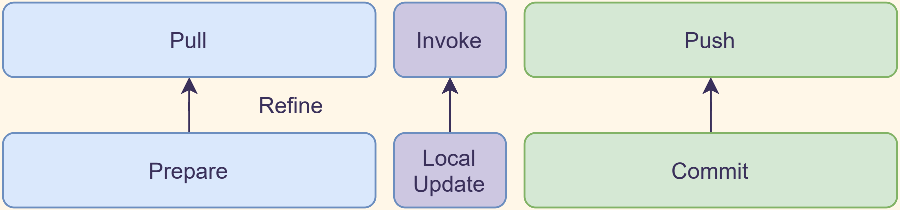
Non-Standard Method Calls

```
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; }
    }
  }
}
```

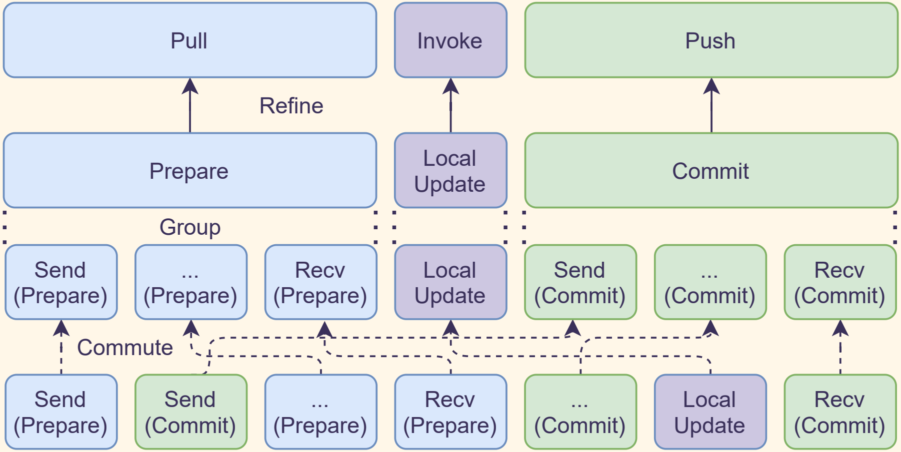
Connection with Distributed Protocols



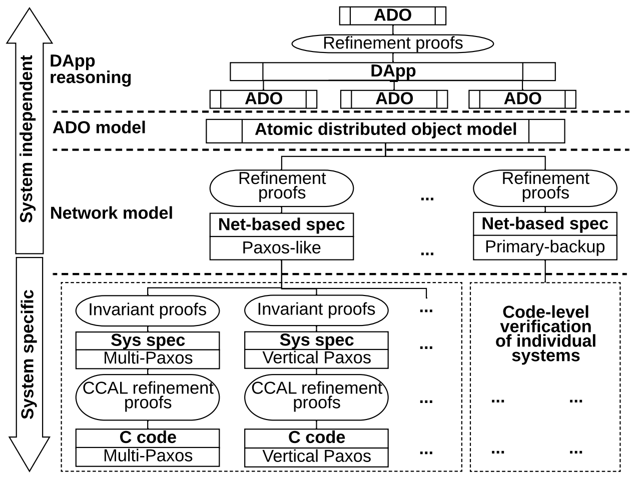
Refinement



Refinement


Pull
Invoke
Push
Refine
Prepare
Local Update
Commit
Group
Send (Prepare)
... (Prepare)
Recv (Prepare)
Local Update
Send (Commit)
... (Commit)
Recv (Commit)
Commute
Send (Prepare)
Send (Commit)
... (Prepare)
Recv (Prepare)
... (Commit)
Local Update
Recv (Commit)

Specification and Proof Effort



Component	LOC
KVLock	646
KVLockFree	359
2PC	559
Paxos-like	5K
Chain Replication	2K
Shared Libraries	11K
Single-Paxos	77
Multi-Paxos	87
Vertical Paxos	97
CASPaxos	78

Code available at `https://zenodo.org/record/5476274`.

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.