Specification & Complexity of Replicated Objects

Hagit Attiya, Technion

Replicated data stores
Geo-distributed systems driving Google, Facebook, Amazon, etc.

This talk

Theoretical exploration of highly-available replicated data stores

- Framework for reasoning
- Results on:
  - Achievable consistency
  - Lower bounds on message size and metadata overhead
- Clarify the landscape

This talk

Theoretical exploration of highly-available replicated data stores

Asynchronous message-passing algorithms implementing shared objects
High availability
Respond without communication

Partition tolerance
Low latency

Propagate Updates Later
Converge to same state
Eventual consistency

CAP Theorem
Cannot provide “strong” consistency (linearizability or serializability)

R1
w:1
r:0
different registers

R2
w:2
r x

w:1, r:0
w:2, r x
Causal Consistency

If an operation is visible, so are its dependencies:

- \( R_1 \) \( w:1 \)
- \( R_2 \) \( r:1, w:3 \)
- \( R_3 \) \( r:3 \)

Exposing Concurrency

How to handle concurrent (conflicting) writes?

- \( R_1 \) \( w:1 \), \( R_2 \) \( w:2, r:\{1,2\} \)

Exposing Concurrency

Practical approach: expose conflicts to user [Dynamo’07]

Multi-valued reg (MVR):
Returns concurrent writes

- \( R_1 \) \( w:1 \)
- \( R_2 \) \( w:2, r:\{1,2\} \)

Exposing Concurrency

Mixing low-level & high-level details

Multi-valued reg (MVR):
Returns concurrent writes

- \( R_1 \) \( w:1 \)
- \( R_2 \) \( w:2, r:\{1,2\} \)
Why Non-Sequential Objects?

Works for single objects
[Perrin, Mostefaoui, Jard’14]

Causality Exposes Concurrency

Avoiding Low-Level Details

Specify replicated objects using visibility in abstract executions
[Burckhardt, Gotsman, Yang, Zawirski]
Abstract Execution
Contains only high-level events

Visibility
Acyclic relation over events, respecting per-replica order

Visibility ≠ message deliveries

Object Specification
Operation’s response is determined by the operations visible to it
Object Specification

Concrete execution implements the object if it complies with an abstract object execution

R1  w:1
R2  w:2 w:3
R3  r:3 r:{1,2}

Eventual consistency

Infinite abstract execution is eventually consistent if an operation is invisible to only finitely many operations

Liveness property

Has the same responses at each replica

[Burckhardt, Gotsman, Yang, Zawirski]

Implies Vogels’ informal definition
Causal consistency

Consistency model: prefix-closed set of abstract executions
Causal consistency: visibility is transitive

Comparing Consistency Models

A consistency model is satisfied when all concrete executions comply with one of its abstract executions
Fewer abstract executions $\Rightarrow$ stronger model
Bayou, PRACTI, COPS... satisfy causal consistency
Can we satisfy a stronger model?

Consistency Limit Result

Theorem: Eventually consistent data store D does not satisfy a consistency model stronger than observable causal consistency (OCC)
OCC hides concurrency unless user can infer it

Deriving OCC

Goal: Comply with abstract execution without concurrency
**Deriving OCC**

**Goal:** Comply with abstract execution without concurrency.

- **R1:** w:1
- **R2:** w:2
- **R3:** r:2

---

**Message lower bound**

- **n:** # of replicas
- **s:** # of MVRs, each of \(~\lg k\) bits

**Theorem:** \(\Omega(\min\{n, s\} \lg k)\) bit message lower bound for causally & eventually consistent data store

- **n:** # of replicas
- **s:** # of MVRs, each of \(~\lg k\) bits

**Tight?**
- Don’t know

**Assumptions**
- invisible reads
- msg sending, real time
Collaborative Text Editing

Collaborative Text Editing: Under the Hood

High availability: Respond without communication

Partition tolerance

Low latency

Fry onions
Collaborative Text Editing: Under the Hood
Eventual consistency: Propagate changes converge to same document

Replicated Object: List
Basic shared document editing operations:

\[
\begin{align*}
\text{ins}(a, \text{pos}) & \quad \text{del}(a) & \quad \text{read}() \\
(\text{inserted elements are unique})
\end{align*}
\]

Every op returns state of the list:

\[
\text{ins}(x, 0) : x \quad \text{ins}(a, 1) : xa
\]

Expected List Behavior

Expected List Behavior

some systems allow this
List Semantics
Shared document editing operations:

\[
\text{ins}(a, \text{pos}) \quad \text{del}(a) \quad \text{read()}
\]
(inserted elements are unique)

Every op returns state of the list

List Semantics

What does *previous* mean?

Can’t use messages received (low-level)
Again, use *visibility* in abstract executions

Every op returns state of the list
List of elements, each with previous *ins()*
but no previous *del()*

Implementing a List Object

Every concrete execution complies with an abstract list execution

R1
\[
\text{ins}(a, k)
\]

R2
\[
\text{ins}(b, k) \quad \text{read}
\]

R3
\[
\text{read} \quad \text{del}(b)
\]
Implementing a List Object

Each operation returns ordered list of elements with visible `ins()` but no visible `del()`

R1: `ins(a,k)`
R2: `ins(b,k)` read
R3: read `del(b)`

Strong List Specification

**Strong list order:** ∃ irreflexive relation that’s transitive & total on all inserted elements

Intuition: remembers deleted elements

```
ins(a,0) : ax  ins(b,1) : xb

ins(x,0) : x

del(x) :

read : ab
```

```
ins(a,0) : ax  ins(b,1) : xb

ins(x,0) : x

del(x) :

read : ab
```

```
ins(a,0) : ax  ins(b,1) : xb

ins(x,0) : x

del(x) :

read : ab
```

```
ins(a,0) : ax  ins(b,1) : xb

ins(x,0) : x

del(x) :

read : ab
```
**Strong List Specification**

*Strong list order:* ∃ irreflexive relation that’s transitive & total on all inserted elements

*Intuition:* remembers deleted elements

![Diagram of Strong List Specification](image)

**Weak List Specification**

*Weak list order:* ∃ irreflexive relation that’s transitive & total on elements returned by an operation

![Diagram of Weak List Specification](image)

**Algorithm for the Strong List**

Replicated Growable Array (RGA)

[Roh, Jeon, Kim, Lee. JPDC 2011]

Resolve order of elements concurrently inserted at the same position with *Timestamped Insertion (TI) Data Structure*

Keep tombstones for deleted elements
**RGA: Timestamped Insertion**

Stores list content & timestamp metadata

To **read**, list elements in **prefix** order, with children appearing in decreasing timestamp order

To **insert** at position $k$, pick a timestamp $>$ than all existing timestamps; insert new node as the **child** of the immediately preceding element

Message: the new node

---

**RGA: Concurrent Insertions**

---

**RGA: Deletions**

To **delete** just mark the element as deleted, leaving a **tombstone**

Change = deletion + insertion

$\Rightarrow$ **Lots of tombstones**

---

**RGA: Deletions**

$\Rightarrow$ **Lots of tombstones**
**Are Tombstones Necessary?**

Some algorithms don’t have them:

- **Treedoc** [Preguiça, Marqués, Shapiro, Letia. ICDCS 2009]
- **Logoot** [Weiss, Urso, Molli. ICDCS 2009]

Element position = sequence of edge labels on the path from the root of the tree

Label stays the same after nodes are deleted

- Operational transformations (OT)
  - Log updates; transform them locally

---

**Metadata Lower Bound**

There is an execution with $D$ deletions, in which a replica has $\Omega(D)$-bit metadata overhead

- Even with **causal** atomic broadcast
- Even for **weak** specification
- **Only for push-based protocols**
  - a replica sends updates to other replicas & merges updates from other replicas into its state **as soon as possible**

---

**Metadata Lower Bound**

There is an execution with $D$ deletions, in which a replica has $\Omega(D)$-bit metadata overhead

Execution in which the list at some replica is “*” but replica’s state is $\Omega(D)$ bits
Proof Technique

There are $2^D$ such strings $\Rightarrow$ for some $w$, size of replica state after $\alpha_w$ is $\Omega(D)$ bits

$\forall D$-bit string $w$, construct an execution $\alpha_w$ s.t.:

✓ A replica performs $D$ deletions and receives no messages
✓ After $\alpha_w$, the list at the replica is “*”
✓ $w$ can be decoded from the state of the replica
✓ The replica has no pending messages

Example: encoding $w = 01$

<table>
<thead>
<tr>
<th>$[0]_0$</th>
<th>send $m_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[1][0]_0$</td>
<td>send $m_2$</td>
</tr>
<tr>
<td>$[1][1][0]_0$</td>
<td>send $m_3$</td>
</tr>
<tr>
<td>$[1][1][2][0]_0$</td>
<td>send $m_4$</td>
</tr>
<tr>
<td>$[1][1][2][*][0]_0$</td>
<td>send $m_5$</td>
</tr>
</tbody>
</table>

Output: encoding replica state, $\sigma$

Encoding $w$

Encode the path to the $w^{th}$ leaf of a complete binary tree

Considering $w+1$ as a number in $[1..2^D]$

✓ After $\alpha_w$, the list at the replica is “*”
✓ $w$ can be decoded from the state of the replica

Decoding $w$ from $\sigma$ (strong spec)

Reconstruct $\alpha_w$ iteratively

<table>
<thead>
<tr>
<th>$[0]_0$</th>
<th>send $m_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[1][0]_0$</td>
<td>send $m_2$</td>
</tr>
<tr>
<td>$[1][1][2][0]_0$</td>
<td>send $m_3$</td>
</tr>
<tr>
<td>$[1][1][2][*][0]_0$</td>
<td>send $m_4$</td>
</tr>
<tr>
<td>$[1][1][2][*][1][0]_0$</td>
<td>send $m_5$</td>
</tr>
</tbody>
</table>
**Decoding $W$ from $\sigma$ (strong spec)**

### R1

- **Decoding $W$ from $\sigma$ (strong spec)**
  - Reconstruct $\alpha_W$ iteratively
  - We know first step in $\alpha_W$

### R2

- **Prefix ending with ins($[i]_i$)**
  - Decode position of $[i+1]_{i+1}$

### R1 @ $\sigma$

- **Read**: $[0]_0$
- **Send**: $m_1$

- **R1 @ $\sigma$**
  - **Read**: $[0]_0$
  - **R2**
    - **Read**: $[0]_0$
  - **R1 @ $\sigma$**
    - **Read**: $[0]_0$

### Decoding $W$ from $\sigma$ (strong spec)

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

### Decoding $W$ from $\sigma$ (strong spec)

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

### Decoding $W$ from $\sigma$ (strong spec)

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

### Decoding $W$ from $\sigma$ (strong spec)

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

---

**Decoding $W$ from $\sigma$ (strong spec)**

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

- **R1 @ $\sigma$**
  - **Read**: $[0]_0$
  - **R2**
    - **Read**: $[0]_0$
  - **R1 @ $\sigma$**
    - **Read**: $[0]_0$

---

**Decoding $W$ from $\sigma$ (strong spec)**

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

- **R1 @ $\sigma$**
  - **Read**: $[0]_0$
  - **R2**
    - **Read**: $[0]_0$
  - **R1 @ $\sigma$**
    - **Read**: $[0]_0$

---

**Decoding $W$ from $\sigma$ (strong spec)**

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

- **R1 @ $\sigma$**
  - **Read**: $[0]_0$
  - **R2**
    - **Read**: $[0]_0$
  - **R1 @ $\sigma$**
    - **Read**: $[0]_0$

---

**Decoding $W$ from $\sigma$ (strong spec)**

- **R1**
  - **Send**: $m_1$
  - **Send**: $m_2$
  - **Send**: $m_3$
  - **Send**: $m_4$
  - **Send**: $m_5$

- **R1 @ $\sigma$**
  - **Read**: $[0]_0$
  - **R2**
    - **Read**: $[0]_0$
  - **R1 @ $\sigma$**
    - **Read**: $[0]_0$
Decoding $\mathcal{W}$ from $\sigma$ (strong spec)

Reconstruct $\alpha_w$ iteratively
- We know first step in $\alpha_w$
- Prefix ending with $\text{ins}([i,i])$
  $\Rightarrow$ decode position of $[i+1]_{i+1}$
- $x^* \Rightarrow \text{ith bit is 1}$
- $*x \Rightarrow \text{ith bit is 0}$

Reconstruct $\alpha_w$ iteratively
- We know first step in $\alpha_w$
- Prefix ending with $\text{ins}([i,i])$
  $\Rightarrow$ decode position of $[i+1]_{i+1}$
- $x^* \Rightarrow \text{ith bit is 1}$
- $*x \Rightarrow \text{ith bit is 0}$

Extension to weak spec

Construction still works!

Extensions
- Result holds for client-server model
  - Proof’s execution satisfies atomic broadcast:
    All replicas receive messages in same order
  - Replicas can maintain server’s state
  - Encoding replica receives no messages = Server is in its initial state
- $\Omega(D)$-bit metadata overhead for clients
Weak Specification

- Result holds also for the weak specification
- Comes from client-server model
- For P2P, equivalent to strong spec?
- Captures real systems?
  Conjecture: Jupiter (Google Docs algorithm)

Wrap Up

Systematic study of replicated data stores

- Tighten consistency result, message size & metadata bounds
- Explore assumptions (push-based): remove them or get better algorithms by violating them
- Incorporate garbage collection
- Go beyond plain text editing, e.g., spreadsheets and other objects

READ MORE ABOUT IT...
