Erasure Coding in Object Stores: Challenges and Opportunities

Lewis Tseng
Boston College

July 2018, PODC
Acknowledgements

Viveck R. Cadambe
Erez Kantor
Alexander Shvartsman

The slide deck is based on the SCNDS tutorial (2017) by Kishori and Prakash Narayana Moorthy
Outline

Motivation

Replication-based algorithm: ABD

Erasure code-based algorithms

Opportunities
Motivation
How do we get here?

MORE THAN 243,000 PHOTOS UPLOADED
MORE THAN 3.8 MILLION SEARCHES ON GOOGLE
MORE THAN 350,000 TWEETS SENT
MORE THAN 65,000 PHOTOS UPLOADED
MORE THAN 210,000 SNAPS UPLOADED
120 NEW ACCOUNTS CREATED ON LINKEDIN
MORE THAN 29 MILLION MESSAGES PROCESSED
1 MILLION PHOTOS
175,000 VIDEO MESSAGES SHARED
MORE THAN 156 MILLION E-MAILS SENT
MORE THAN 87,000 HOURS OF VIDEO WATCHED
MORE THAN 800,000 FILES UPLOADED ON DROPBOX
MORE THAN 700,000 HOURS OF VIDEO WATCHED
MORE THAN 5,500 CHECKINS ON FOURSQUARE
MORE THAN 2,000,000 MINUTES OF CALLS DONE BY SKYPE USERS
MORE THAN 200 EVENT TICKETS SOLD ON EVENTBRITE
MORE THAN 1,000 IMAGES UPLOADED
MORE THAN 50 NEW REVIEWS
MORE THAN 1,000,000 SWIPES
MORE THAN 500,000 APPS DOWNLOADED
18,000 MATCHES ON TINDER
16,550 VIDEO VIEWS ON YOUTUBE
200 MORE EVENTS ON EVENTBRITE
40 MORE PHOTOS ON IMAGUR
60 MORE THINGS ON NETFLIX
How do we get here?

Internet
Mobile computing
Big data
• Storage
• Processing
Key-Value Storage

Clients  write $V$ to key $K$:  $W(K, V)$
read $V$ from key $K$:  $V=R(K)$

Application:  Geo-replicated datacenters
Storage Service

Asynchrony
Distributed
Fault-tolerant

Internet

Clients
Geo-Replication

[W(K,0)]

[K=0]

[Lloyd et al. ‘11]
Geo-Replication

[Image of a map with annotations]

W(K,0)  R(K)  K=0

[Lloyd et al. ‘11]
Key-Value Storage for Unstructured Data

Source: http://bigdata.black/infrastructure/storage/unstructured-data/
Explosion of (Unstructured) Data

The Cambrian Explosion...of Data

Exabytes (billions of GB)

Key-Value Storage in Industry

• Amazon Dynamo

• Apache Cassandra

• Google Spanner

• CockroachDB, HBase, MongoDB, ... many more
Key-Value Storage

• Latency
• Throughput

“Latency matters. Amazon found every 100ms of latency cost them 1% in sales. Google found an extra 0.5 seconds in search page generation time dropped traffic by 20%.” (article from 2008!)

Source: https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/
Key-Value Storage

• Latency
• Throughput
• Consistency
Issue of Consistency

K = 0 or 1?

Need ordering of ops

Datacenter A

Datacenter B

K = 0

K = 1

K = 0

K = 1
Consistency Models

Atomicity: $X = 1$  
[Herlihy, Wing ‘90]

Eventual: $X = 0$ or $1$ or NULL  
[DeCandia et al. ‘07]
A Semi-Formal Definition of Atomicity

- Each read operation returns the value of the preceding write operation that “completed”, such that
- This value is at least as recent as that returned by any preceding read
Each read operation returns the value of the preceding write operation that “completed”, such that

• This value is at least as recent as that returned by any preceding read

Imagine as though the clients’ ops occurred “instantaneously” at the red serialization point.
A Semi-Formal Definition of Atomicity

- Each read operation returns the value of the preceding write operation that “completed”, such that
- This value is at least as recent as that returned by any preceding read

<table>
<thead>
<tr>
<th></th>
<th>$v_1$</th>
<th>$v_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read 1</td>
<td></td>
<td>$v_1$</td>
</tr>
<tr>
<td>Read 2</td>
<td></td>
<td>$v_2$</td>
</tr>
<tr>
<td>Read 3</td>
<td></td>
<td>$v_1$</td>
</tr>
</tbody>
</table>

global time
A Semi-Formal Definition of Atomicity

- Each read operation returns the value of the preceding write operation that “completed”, such that
- This value is at least as recent as that returned by any preceding read

![Diagram showing the sequence of writes and reads with timestamps]

- Writes: $v_1$ * $v_2$
- Read 1: * $v_1$
- Read 2: * $v_2$
- Read 3: * $v_1$

(global time)
A Semi-Formal Definition of Atomicity

- Each read operation returns the value of the preceding write operation that “completed”, such that
- This value is at least as recent as that returned by any preceding read

<table>
<thead>
<tr>
<th>Writes</th>
<th>Read 1</th>
<th>Read 2</th>
<th>Read 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>*</td>
<td></td>
<td>$v_1$</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>$v_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$v_1$</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$v_2$</td>
<td>$v_1$</td>
</tr>
</tbody>
</table>
This Talk: Focus on Atomicity

• Linearizability (or “strong consistency”)

• Ease-of-use

• Composability (or locality)
  • If operations on each object are linearizable, then all operations on all objects are linearizable
Key-Value Storage

• Latency
• Throughput
• Consistency
• Storage cost!
Storage Cost?

This Talk: Goal

• Atomic key-value storage (or object store)

• Focus: saving storage & communication cost
Replication vs. Coding
Replicated Storage

Fault-tolerance
Low latency
Replicated Storage

Fault-tolerance
Low latency
Replicated Storage

Fault-tolerance
Low latency

n copies of data
• Storage cost: n
• Comm. cost: n
Replication vs. Coding-based
Erasure Codes: \([n, k]\) MDS Codes

Value \( \mathcal{U} \) > Divide into \( k \) equal parts

\( v_1 \quad v_2 \quad v_3 \) \quad \ldots \quad \ldots \quad \ldots \quad v_k 

Encode
Erasure Codes: \([n, k]\) MDS Codes

Value

\[ V \]

Divide into \(k\) equal parts

\[ v_1 \quad v_2 \quad v_3 \quad \cdots \quad v_k \]

Encode

Codeword

\[ c_1 \quad c_2 \quad \cdots \quad c_n \]

(each encoded part is the same size as original parts)
Erasure Codes: \([n, k]\) MDS Codes

Value

\[ U \]

Divide into \( k \) equal parts

\[ v_1 \quad v_2 \quad v_3 \quad \ldots \quad v_k \]

Encode

Codeword

\[ c_1 \quad c_2 \quad \ldots \quad \ldots \quad \ldots \quad c_n \]

Servers

\[ S_1 \quad S_2 \quad \ldots \quad \ldots \quad \ldots \quad S_n \]
Erasure Codes: \([n, k]\) MDS Codes

- **Value:** Divide into \(k\) equal parts
- **Codeword:** Encode
- **Servers:**
  - \(S_1, S_2, \ldots, S_n\)
- **Recovered value:** (any \(k\) coded elements suffice)
Erasure Codes: \([n, k]\) MDS Codes

- **Value**
  - Divide into \(k\) equal parts
  - \(v_1, v_2, \ldots, v_k\)

- **Codeword**
  - \(c_1, c_2, \ldots, c_n\)

- **Servers**
  - \(S_1, S_2, \ldots, S_n\)

- **Recovered value**
  - (any \(k\) coded elements suffice)

Tolerate crashes of any \((n - k)\) nodes
Previous scheme sufficient?

• Yes for cold storage
  write once, and read it infrequently

• No for hot storage
  concurrent write and read operations
Erasure Coding operation is slow

Zhang et. al. “Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication”, USENIX FAST 2016

... For example, a single Intel Xeon E3-1230v3 CPU core can encode data at 5.26GB/s for Reed-Solomon(3,5) codes, which is faster than even current high-end NIC with 40Gb/s bandwidth...
Erasure Codes: Common Doubts

Erasure Coding operation is slow

Zhang et. al. “Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication”, USENIX FAST 2016

… For example, a single Intel Xeon E3-1230v3 CPU core can encode data at 5.26GB/s for Reed-Solomon(3,5) codes, which is faster than even current high-end NIC with 40Gb/s bandwidth…

Erasure Coding only works for large object

Chen et. al. “Giza: Erasure Coding Objects across Global Data Centers”, USENIX ATC 2017

We observe that less than 0.9% of the total storage capacity is occupied by objects smaller than 4MB. This suggests that, to optimize storage cost, it is sufficient for Giza to focus on objects of 4MB and larger…
Two Recent Systems in Industry

• **Microsoft Giza [USENIX ATC ‘17]**
  • Guarantees strong consistency
  • Supports erasure-code across data-centers
  • Tolerates 1 failure
  • Uses Paxos and Fast-Paxos

• **OpenStack Swift**
  • Powers large object storage clouds
  • Is used by AT&T, Intel, NASA, Walmart, Yahoo!, etc.
  • Guarantees eventual consistency
  • Supports erasure-code
Replication-based Algorithm: ABD

## Desirable Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>Ease-of-use</td>
</tr>
<tr>
<td>Liveness</td>
<td>An operation completes</td>
</tr>
<tr>
<td>Fault-tolerance</td>
<td>Crash failures of a certain fraction of servers and any clients</td>
</tr>
<tr>
<td>Storage cost</td>
<td>Low storage overhead</td>
</tr>
<tr>
<td>Comm. cost</td>
<td># bits in comm.</td>
</tr>
<tr>
<td>Latency</td>
<td># rounds</td>
</tr>
</tbody>
</table>

43
# Replication method: Consensus vs. Leaderless

<table>
<thead>
<tr>
<th></th>
<th>Consensus</th>
<th>Leaderless Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Suitable scenario</strong></td>
<td>Very few writes per object</td>
<td>Any amount of concurrent write</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>One-round operation if no write-concurrency</td>
<td>Typically two or more rounds</td>
</tr>
<tr>
<td><strong>Liveness</strong></td>
<td>Slow client operations during high write-concurrency</td>
<td>Strong liveness guarantees</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>HBase, Microsoft Giza</td>
<td>Amazon Dynamo, Apache Cassandra, ABD</td>
</tr>
</tbody>
</table>
## Replication method: Consensus vs. Leaderless

<table>
<thead>
<tr>
<th></th>
<th>Consensus</th>
<th>Leaderless Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Suitable scenario</strong></td>
<td>Very few writes per object</td>
<td>Any amount of concurrent write</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>One-round operation if no write-concurrency</td>
<td>Typically two or more rounds</td>
</tr>
<tr>
<td><strong>Liveness</strong></td>
<td>Slow client operations during high write-concurrency</td>
<td>Strong liveness guarantees</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>HBase, Microsoft Giza</td>
<td>Amazon Dynamo, Apache Cassandra, ABD</td>
</tr>
</tbody>
</table>

This Talk focuses on Leaderless method
ABD: Leaderless Method

Write and read from a majority of servers
• write or read quorum

Intersection of any quorums: at least one server
• key property for leaderless-based method

Fault-tolerance
• a minority of servers fail
Write Quorum
Quorum Intersection: Write & Read Quorum

Writer

ACK

Reader
First Challenge

How to get timestamp for concurrent writes?
First Challenge

How to get timestamp for concurrent writes?

Use Quorum!

Writer needs to read first!
Writer: First Phase

Tag: (ts, writer-ID)

Server: return largest known tag

Property: learn the most recent tag
ABD: Pseudo-Code

Writer:
• Acquire latest tag via get-tag; Send incremented tagged value to server; Return after majority ACKs

Reader:
• Send read-request; Wait for majority ACKs; Send latest value to server; Return latest value after receiving majority ACKs

Server:
• Respond to query with tag; Store latest value from client; Send ACK
• Respond to read request with value
Writer & Server

Writer(∪)

(tag, value)

ACK

(t_1, v) → (t_0, v_0) → (t_1, v)

(t_1, v) → (t_0, v_0) → (t_1, v)

(t_1, v) → (t_0, v_0) → (t_1, v)

(t_1, v) → (t_0, v_0) → (t_1, v)

(t_1, v) → (t_0, v_0) → (t_1, v)
Reader: Get the most recent value!
Second Challenge: Concurrent Write

**Diagram:**

- **Writer:**
  - $(t_0, v_0)$
  - $(t_1, v)$

- **Reader 1:**
  - $(t_1, v)$

- **Reader 2:**
  - $(t_0, v_0)$

- **Global Time:**
Second Challenge: Concurrent Write
Second Challenge: Concurrent Write

Writer(\(U\))

\((t_1, v)\)

\((t_1, v)\)  Ack

\((t_0, v_0)\)  \((t_1, v)\)

\((t_0, v_0)\)  \((t_0, v_0)\)

\((t_1, v)\)

\((t_0, v_0)\)  \((t_1, v)\)

\((t_0, v_0)\)  \((t_0, v_0)\)

\((t_1, v)\)

Reader

Reader 2
Second Challenge: Concurrent Write

Reader 1 needs to update the new value (Write-back technique)
ABD: Pseudo-Code

**Writer:**
- Acquire latest tag via get-tag;  
  Send incremented tagged value to server;  
  Return after majority ACKs

**Reader:**
- Send read-request;  
  Wait for majority ACKs;  
  Send latest value to server;  
  Return latest value after receiving majority ACKs

**Server:**
- Respond to query with tag;  
  Store latest value from client;  
  Send ACK
- Respond to read request with value
ABD: Atomicity

Each read returns the value of the preceding write

Each read’s value is at least as recent as that returned by any preceding read
ABD: Atomicity

Each read returns the value of the preceding write

Write Quorum  Read Quorum  Quorum Intersection

Each read’s value is at least as recent as that returned by any preceding read

Write-back
ABD: Performance

- Assume Value $\nu$ has size 1
- Comm. cost: total amount of data transferred in the worst case
- Ignore overhead of tags

<table>
<thead>
<tr>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>2n</td>
<td>n</td>
<td>n/2 - 1</td>
</tr>
</tbody>
</table>
ABD: Performance

- Assume Value $v$ has size 1
- Comm. cost: total amount of data transferred in the worst case
- Ignore overhead of tags

<table>
<thead>
<tr>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$2n$</td>
<td>$n$</td>
<td>$n/2 - 1$</td>
</tr>
</tbody>
</table>

Goal: use Erasure Codes to reduce the cost
CASGC: Coded Atomic Storage-Garbage Collection

[Cadambe-Lynch-Medard-Musial, IEEE NCA 2014]
Extended version in Distributed Computing 2016.
First Observation

Each read needs enough coded elements!

Quorum Intersection
- One element is **not** enough
- You need *k* element
First Observation

Each read needs enough coded elements!

Quorum Intersection
- One element is **not** enough
- You need k element
- \([n, k]\) code
- \(k = n - 2f\)
- \((n+k)/2 \Rightarrow\) quorum set
First Observation

Each read needs enough coded elements!

Quorum Intersection
- One element is **not** enough
- You need \( k \) element
- \([n, k]\) code
- \( k = n - 2f \)
- \( \frac{n+k}{2} \) \(\rightarrow\) quorum set

Write Quorum: \( \frac{n+k}{2} \)
Read Quorum: \( \frac{n+k}{2} \)

Intersection: \( k-1 \)

Total #server: 
\[ n+k - (k-1) = n+1 \]
Specific Challenge: Concurrent Writes

- Write concurrent with Read
- Reader potentially gets coded elements corresponding to different tags

How to ensure Liveness of Read Operations (Decodability)?
Specific Challenge: Concurrent Writes

- Write concurrent with Read
- Reader potentially gets coded elements corresponding to different tags

How to ensure Liveness of Read Operations (Decodability)?
Reveal coded elements to readers only when the original value can be decoded
Tag: Extra Field

(timestamp, writer-ID, fin) ➔ this version is decodable

(timestamp, writer-ID, ___) ➔ this version is ?
CAS: Pseudo-Code

Writer:
• Acquire latest tag via get-tag;  Send write-msg: tag + coded elements; Wait for ACKs from quorum set;
• Send fin-msg;  Wait ACKs from quorum set

Server:
• Upon receiving get-tag:  Respond with latest finalized tag
• Upon receiving write-msg:  Store coded element from client; Send ACK
• Upon receiving fin-msg:  Set fin tag; Send ACK
CAS: Pseudo-Code

Reader:
• Acquire latest tag via get-tag;  Wait for ACKs from quorum set
• Send read-msg with latest tag;  Wait for coded elements from k servers
• Decode and return the value

Server:
• Upon receiving get-tag:  Respond with latest finalized tag
• Upon receiving read-msg:  Set fin tag;
  Respond with coded element if available
CAS: Property

“fin” label: a sufficient number of coded elements (decodable)

Additional write phase: tells servers that elements have stored in a quorum, i.e., decodable
CAS: Property

“fin” label: a sufficient number of coded elements (decodable)

Additional write phase: tells servers that elements have stored in a quorum, i.e., decodable

Servers store all the history
Key Parameter $d$: maximum \#Writes that are concurrent with any Read
GC: First Try

Server: store at most $d+1$ most recent coded elements, discard the rest
GC: First Try

Server: store at most $d+1$ most recent coded elements, discard the rest

Crashed writers will break garbage collection
• Intuitively, these crashed writes are concurrent with all the future reads!
CASGC

Same client code

Server:

• store d+1 most recent coded elements with fin label and all elements in between
• discard the rest
• use gossip to “recover” from the failed write or “detect” the write is not recoverable (i.e., original value not decodable)
Performance

- Assume value $\nu$ has size 1

<table>
<thead>
<tr>
<th></th>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABD</td>
<td>n</td>
<td>2n</td>
<td>n</td>
<td>n/2 - 1</td>
</tr>
<tr>
<td>CASGC</td>
<td>$(n/(n-2f))(d+1)$</td>
<td>$n/(n-2f)$</td>
<td>$n/(n-2f)$</td>
<td>f</td>
</tr>
</tbody>
</table>
## Performance

- Assume value $v$ has size 1

<table>
<thead>
<tr>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABD</strong></td>
<td>$n$</td>
<td>$2n$</td>
<td>$n/2 - 1$</td>
</tr>
<tr>
<td><strong>CASGC</strong></td>
<td>$(n/(n-2f))(d+1)$</td>
<td>$n/(n-2f)$</td>
<td>$n/(n-2f)$</td>
</tr>
</tbody>
</table>

- $[n, k=n-2f]$ code
- Size of one coded element = $1/(n-2f)$
## Performance

- Assume value $v$ has size 1

<table>
<thead>
<tr>
<th></th>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABD</td>
<td>$n$</td>
<td>$2n$</td>
<td>$n$</td>
<td>$n/2 - 1$</td>
</tr>
<tr>
<td>CASGC</td>
<td>$n/2 (d+1)$</td>
<td>$n/2$</td>
<td>$n/2$</td>
<td>$n/2-1$</td>
</tr>
<tr>
<td>CASGC</td>
<td>$(n/(n-2f))(d+1)$</td>
<td>$n/(n-2f)$</td>
<td>$n/(n-2f)$</td>
<td>$f$</td>
</tr>
</tbody>
</table>
Theoretical Results

• [Spiegelman et al. DISC ‘16]
asymptotic storage cost is either $O(f)$ or $O(d)$

• [Berger et al. DISC ‘18]
disintegrated storage: Byz. storage or coded storage
  storage cost: inherently exponential in the size of written values or linear in # readers
Recent Practical Systems

Storage cost: SODA
  [Konwar et al. IPDPS ‘16]

Online repair of crashed nodes: RADON
  [Konwar et al. OPODIS ‘16]

Layered architecture, e.g., proxy, edge servers: LDS
  [Konwar et al. PODC ‘17]

Reconfiguration using consensus: ARES
  [Konwar et al. under submission]
<table>
<thead>
<tr>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permanent (Stable) Cost</td>
<td>Transient Cost</td>
<td>Transient Cost</td>
</tr>
<tr>
<td>Most Important</td>
<td>Second Most Important</td>
<td>Least Important (#Reads &gt;&gt; # Writes)</td>
</tr>
</tbody>
</table>
### SODA

<table>
<thead>
<tr>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permanent (Stable) Cost</td>
<td>Transient Cost</td>
<td>Transient Cost</td>
</tr>
<tr>
<td>Most Important</td>
<td>Second Most Important</td>
<td>Least Important (#Reads &gt;&gt; # Writes)</td>
</tr>
</tbody>
</table>

- Use \([n, k]\) MDS codes
- Tolerate any \((n-k)\) crashes, i.e., \(k = n-f\)
- Sacrifice comm. cost to improve storage cost
## Performance

Assume value $\nu$ has size 1

<table>
<thead>
<tr>
<th></th>
<th>Storage Cost</th>
<th>Read Comm. Cost</th>
<th>Write Comm. Cost</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABD</td>
<td>$n$</td>
<td>$2n$</td>
<td>$n$</td>
<td>$n/2 - 1$</td>
</tr>
<tr>
<td>CASGC</td>
<td>$n/2 (d+1)$</td>
<td>$n/2$</td>
<td>$n/2$</td>
<td>$n/2 - 1$</td>
</tr>
<tr>
<td>SODA</td>
<td>$\leq 2$</td>
<td>$\leq 2 (d+1)$</td>
<td>$\approx n^2$</td>
<td>$n/2 - 1$</td>
</tr>
</tbody>
</table>
SODA: Write

- \([n, k = n - f]\) MDS code
- Only one coded element per node is stored at any time
- Goa: Send the coded elements to all nodes, despite writer crash

Writer \(v\)

- Writer obtains tag as in ABD
- Writer sends \(v\) to nodes 1 to \(f+1\)

Node \(i\) forwards \(v\) to nodes \(i+1\) to \(f+1\)

Node \(i\) computes and sends coded elements for the remaining nodes (also stores its own coded element)
SODA: Write

- \([n, k = n - f]\) MDS code
- Only one coded element per node is stored at any time
- Goa: Send the coded elements to all nodes, despite writer crash

Use servers to forward: handle failed writer

- Writer obtains tag as in ABD
- Writer sends \(v\) to nodes 1 to \(f+1\)

Node \(i\) forwards \(v\) to nodes \(i+1\) to \(f+1\)

Node \(i\) computes and sends coded elements for the remaining nodes (also stores its own coded element)
SODA: Read

• Goal: handle concurrent writes
• Nodes act as relays for the various concurrent writes, until a valid code-word is decoded

  - Select max tag from a majority of nodes
  - Send the max tag $t_1$ back to all nodes

• Every node relays any incoming coded element with tag $t \geq t_1$
• Continue until reader acknowledges read-complete
• Gossip mechanism to stop perpetual relaying (to handle failed readers)
Summary

We need distributed KV storage and it’s important to reduce storage cost

Replication-based algorithm: ABD

Erasure code-based algorithms: CASGC & SODA
Opportunities: Theoretical Works

• Byzantine coded storage
• Reconfiguration without using consensus
• Heterogenous servers & network bandwidth
• Client interaction
Opportunities: Practical Systems

• modular components

• comparison/integration with real-world systems

• one-phase read & write + storage cost one: what consistency can you achieve?

• what if you don’t have reliable communication?
Thanks!

lewis.tseng@bc.edu