ARES: Adaptive, Reconfigurable, Erasure coded atomic Storage

Kishori M Konwar

Joint work with
Nicolas Nicolaou, Viveck Cadambe, Prakash Nayarana Moorthy, Nancy Lynch and Muriel Medard

Jul 7, 2019
ICDCS 2019

algolysis
algorithmic solutions

ΙΔΡΥΜΑ ΕΡΕΥΝΑΣ ΚΑΙ ΚΑΙΝΟΤΟΜΙΑΣ

European Union
European Regional Development Fund

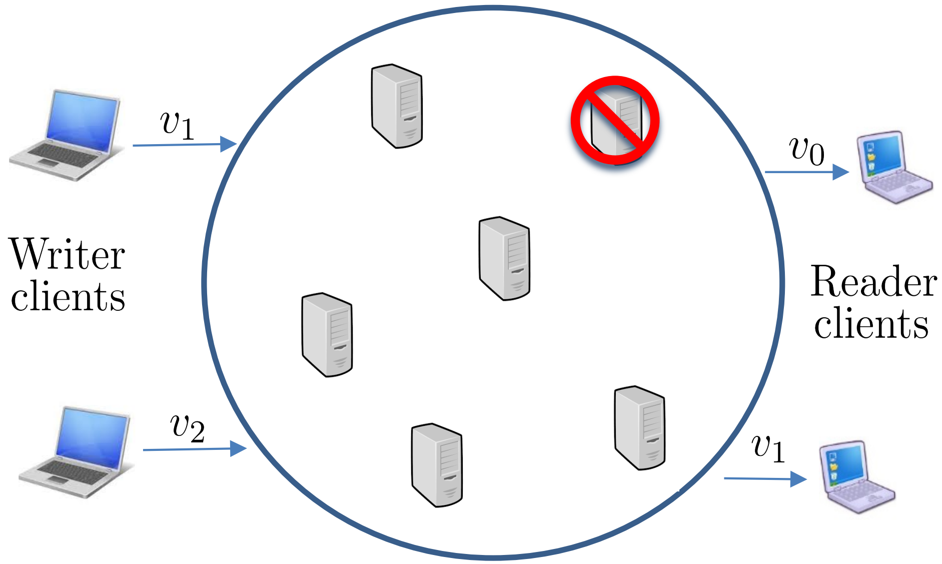
Republic of Cyprus

Structural Funds
of the European Union in Cyprus

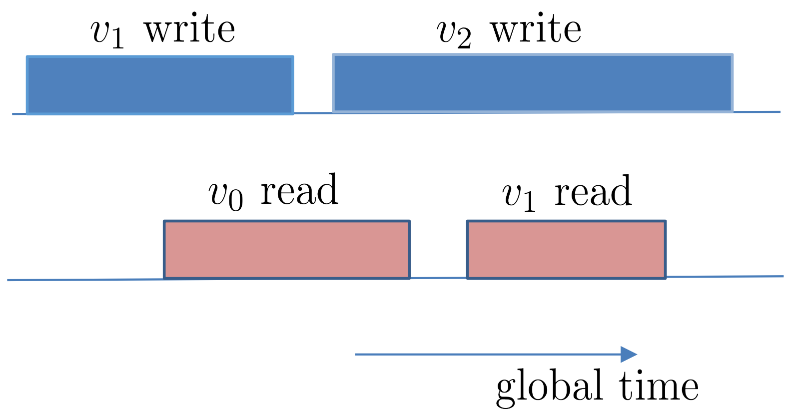
Outline of the Talk

- Distributed Storage – Problem Statement
- System Model
  - Atomicity, Erasure Codes, and Configurations
- DAP: Data Access Primitives
- Reconfiguration Service
  - Configuration Sequence
- Erasure Coded DAP implementation

Problem Statement



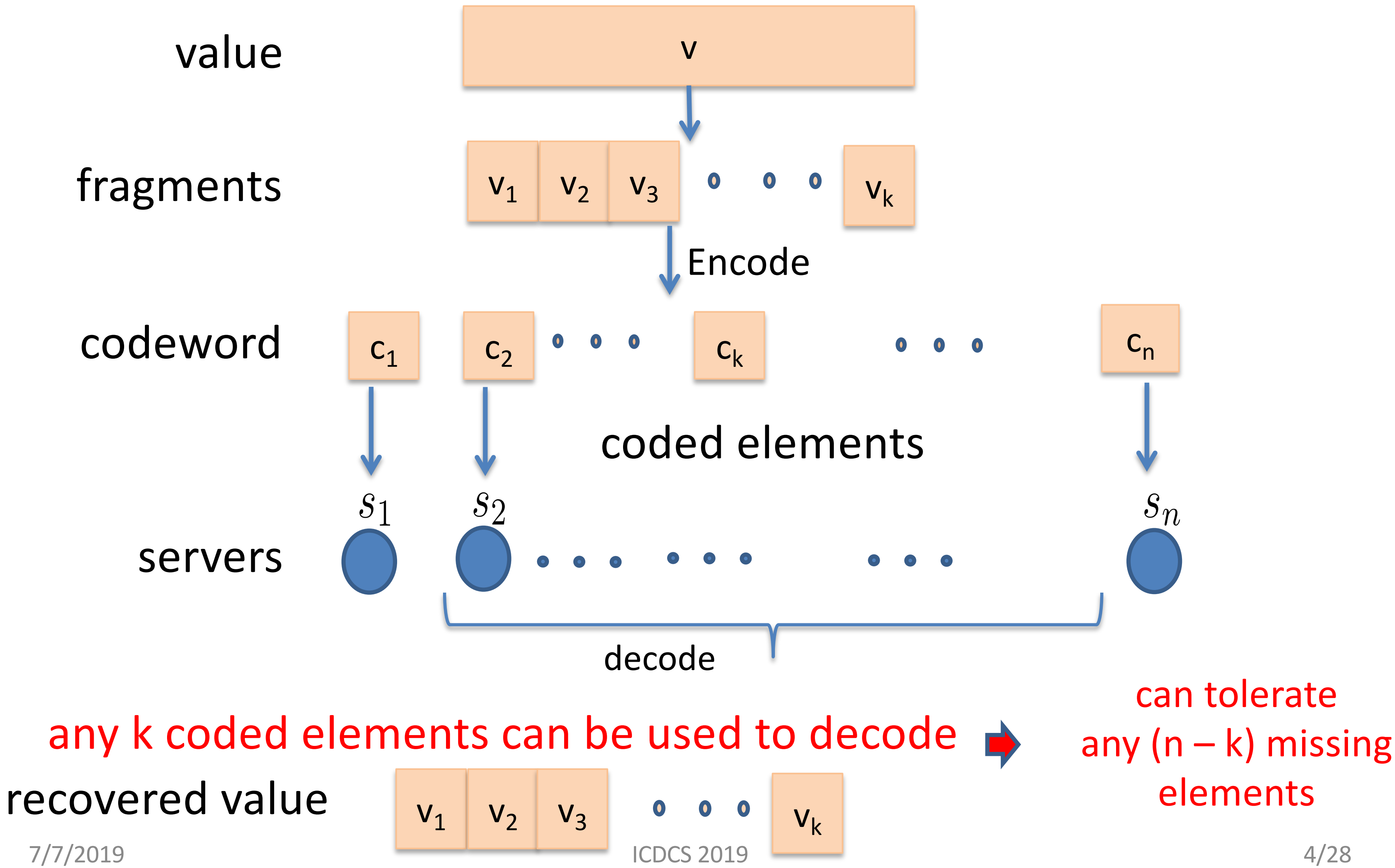
Shared read/write storage object



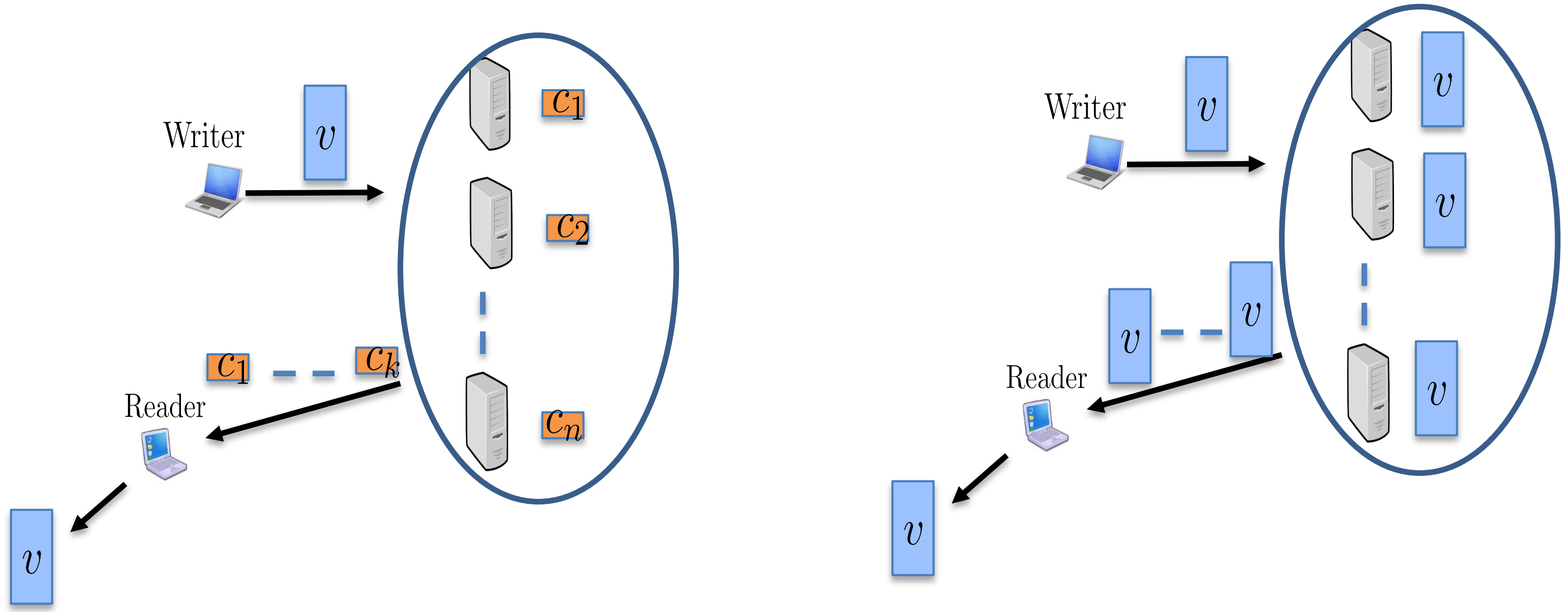
Implementing a fault-tolerant, dynamic shared storage object in an asynchronous, message-passing environment:

- Availability + Survivability => use redundancy
- Asynchrony + Redundancy => concurrent operations
- Behavior of concurrent operations => consistency semantics
  - Safety, Regularity, Atomicity [Lamport86]
- Service Liveness Despite Failures => host reconfiguration

Redundancy: Erasure Codes
([n, k] MDS Codes)



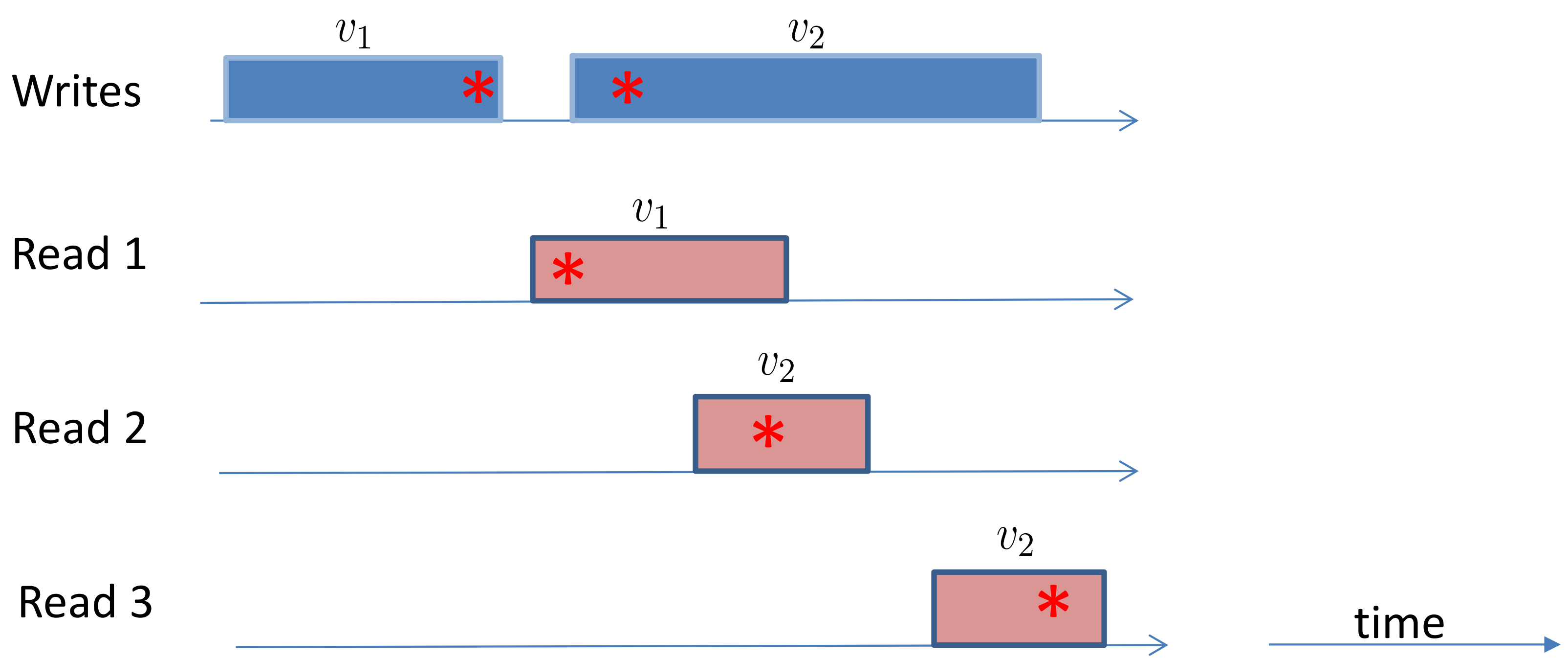
Erasure Code vs Replication



A well-designed algorithm has great potential to reduce storage and communication costs while using erasure codes

Consistency: Atomicity

- Provides the illusion that operations happen in a sequential order
  - a read returns the value of the preceding write
  - a read returns a value at least as recent as that returned by any preceding read



System Model: Definitions

Components

- Clients: W writers & R readers (MWMR)
- Reconfigurers: G recon clients
- Servers: S replica hosts

Operations

- write(*v*): updates the object value to *v*
- read(): retrieves the object value
- reconfig(c): installs a new configuration

Communication

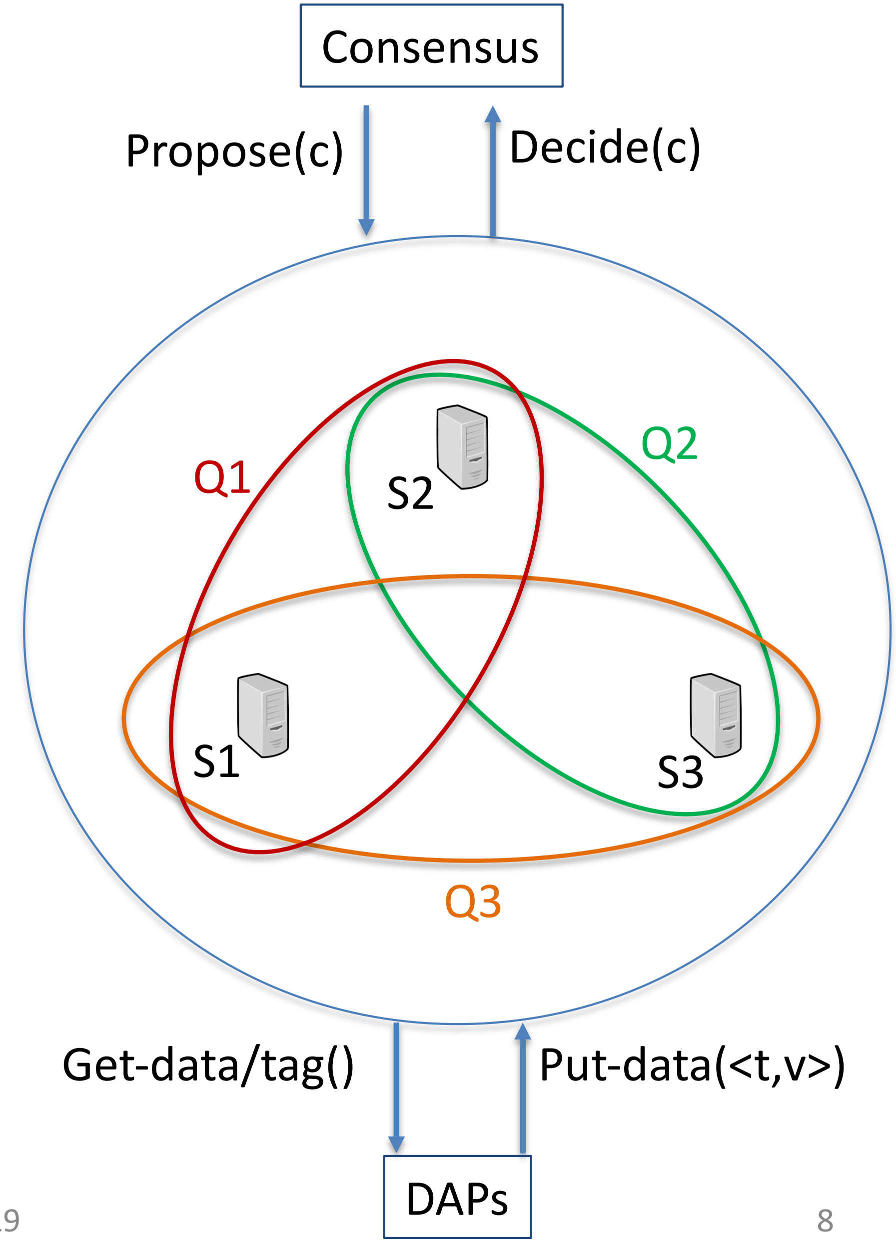
- Asynchronous
- Message-Passing
- Reliable Channels (messages are not lost or altered)

Failures

- Crashes
- Any reader, writer, or recon client
- Server failure specified per configuration

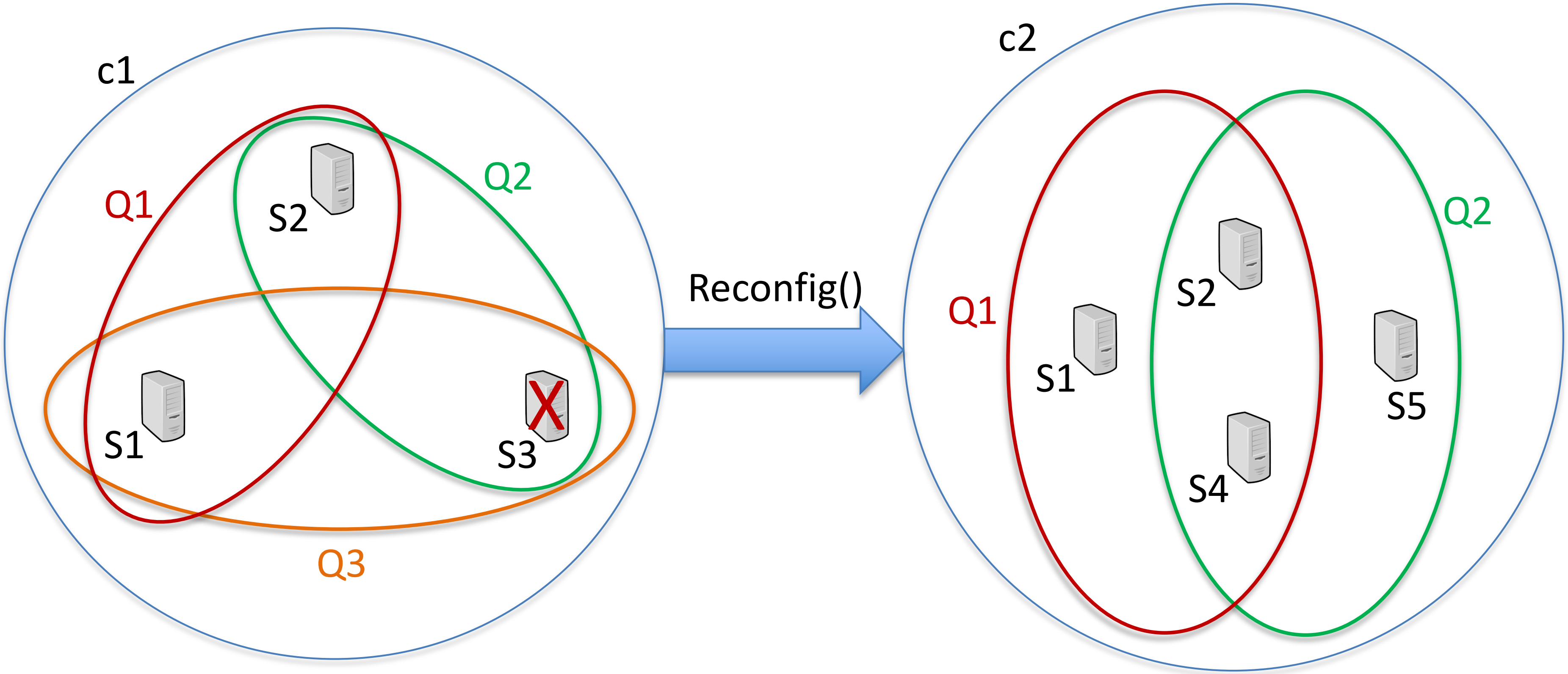
Configurations

- A configuration c is characterized by:
  - A set of servers
  - A quorum set system on servers
  - A consensus instance
  - A DAP implementation

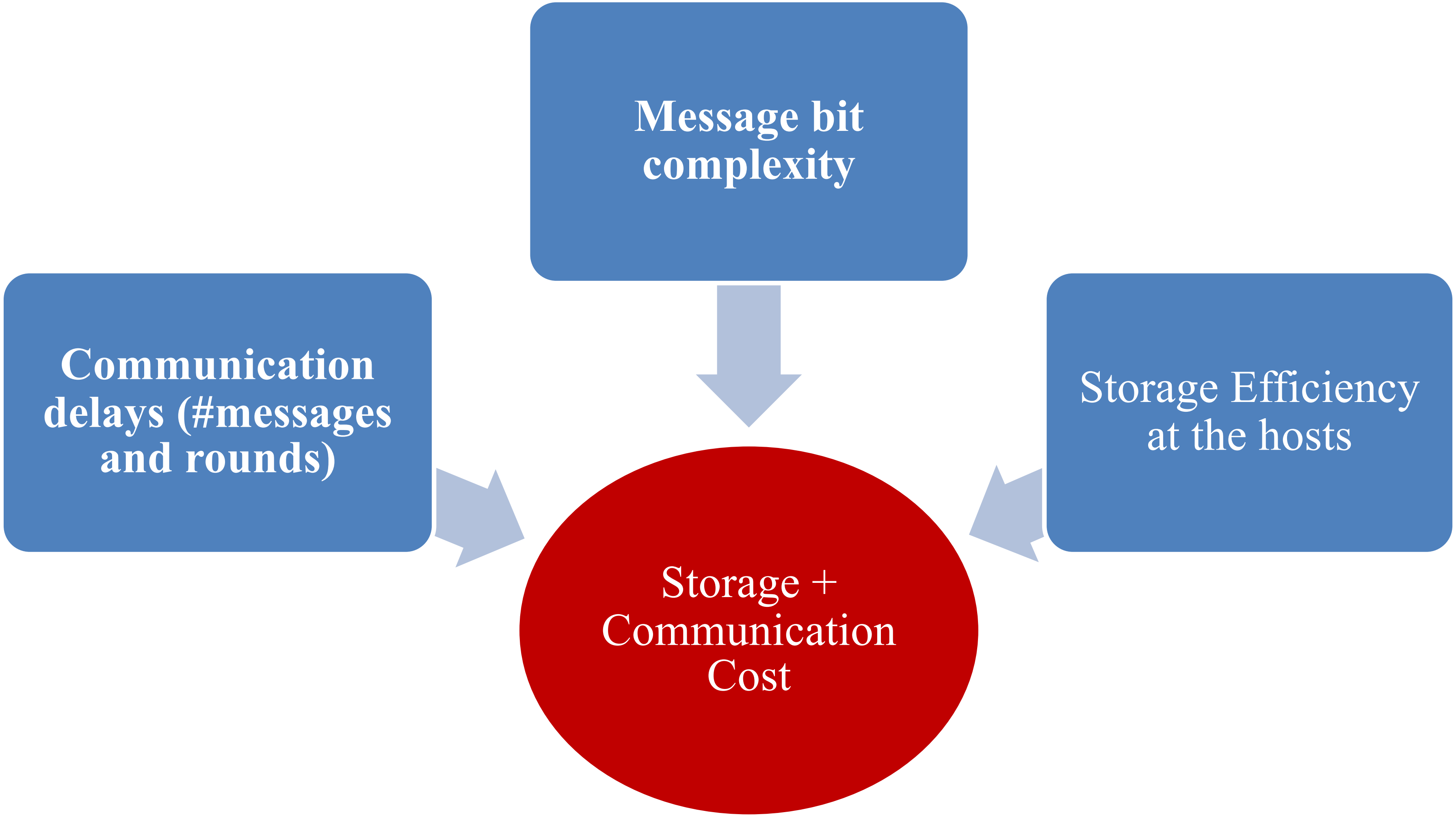


Re-Configuration Operation

- Change the configuration parameters (install new config)
  - Due to failures
  - Due to admin maintenance



Complexity Measure



DAPs: Data Access Primitives

- Operation Ordering: logical tags t = <z,wid>
  - Compared Alphanumerically
- DAP: Building blocks to query/alter tags and data
- For a configuration *c*, any client process p may invoke any of the following data access primitives:

D1. c.get-tag(): *returns a tag* $\tau \in \mathcal{T}$

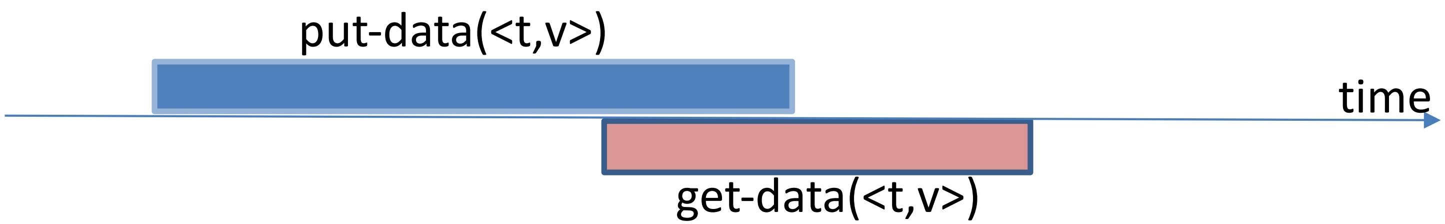
D2. c.get-data(): *returns a tag-value pair* $(\tau, v) \in \mathcal{T} \times \mathcal{V}$

D3. c.put-data($\langle \tau, v \rangle$): *the tag-value pair* $(\tau, v) \in \mathcal{T} \times \mathcal{V}$ *as argument*

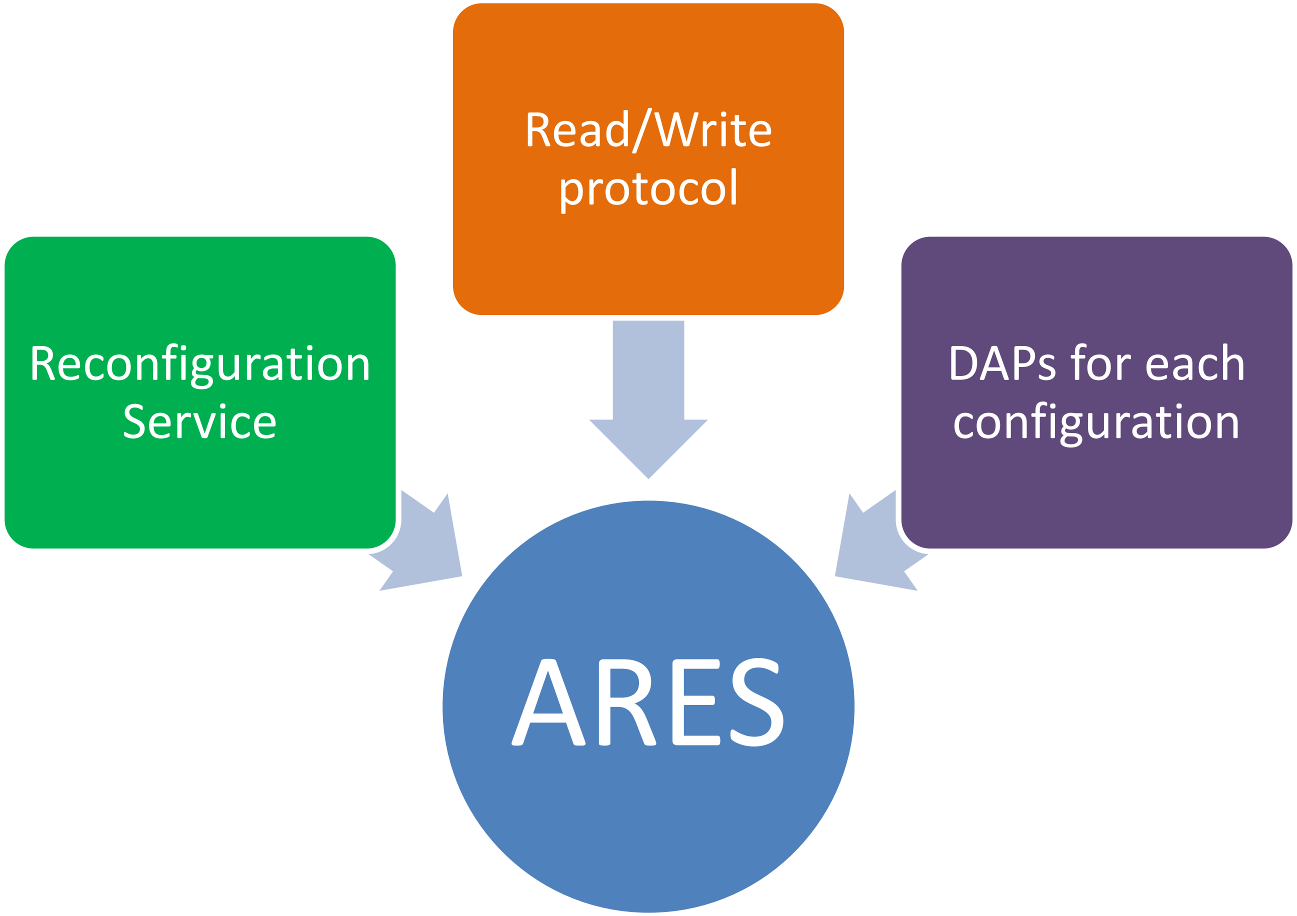
DAP Consistency Properties

- DAPs may be used to yield Atomic Register Implementations if they satisfy the following properties:
  - C1: If a put-data(<t,v>) completes before get-tag/get-data() operation in the same configuration c => get-*() op returns a tag ≥ t

  - C2: if get-data() that returns <t,v> then put-data(<t,v>) completed before or is concurrent to get-data(), else <t,v> = $<t_0, v_0>$



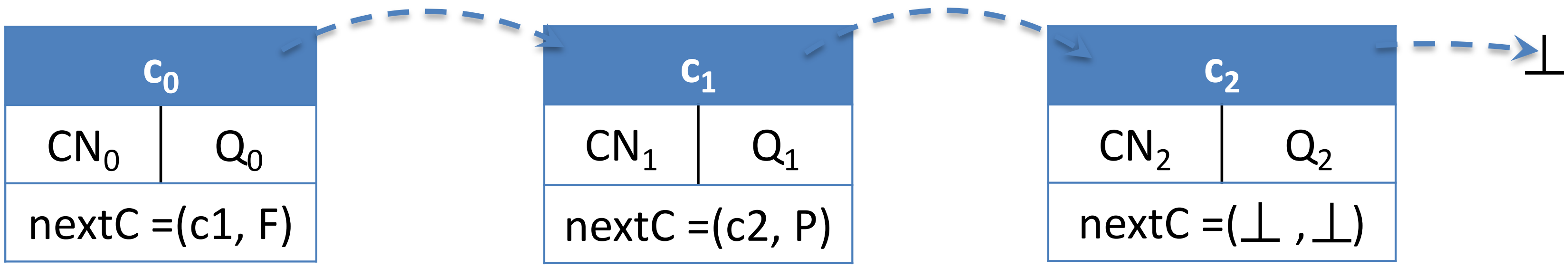
ARES Protocol



DAPs are used by all read/write and reconfig operations

Configuration Sequence

- Global configuration sequence G_L
- Flags {P, F}: pending, finalized
  - Pending: not yet a quorum of servers received msgs
  - Finalized: new configuration propagated to a quorum of servers
- nextC: each server points to the next configuration
  - Same nextC to all servers of a single config c (due to consensus)



Reconfiguration Service

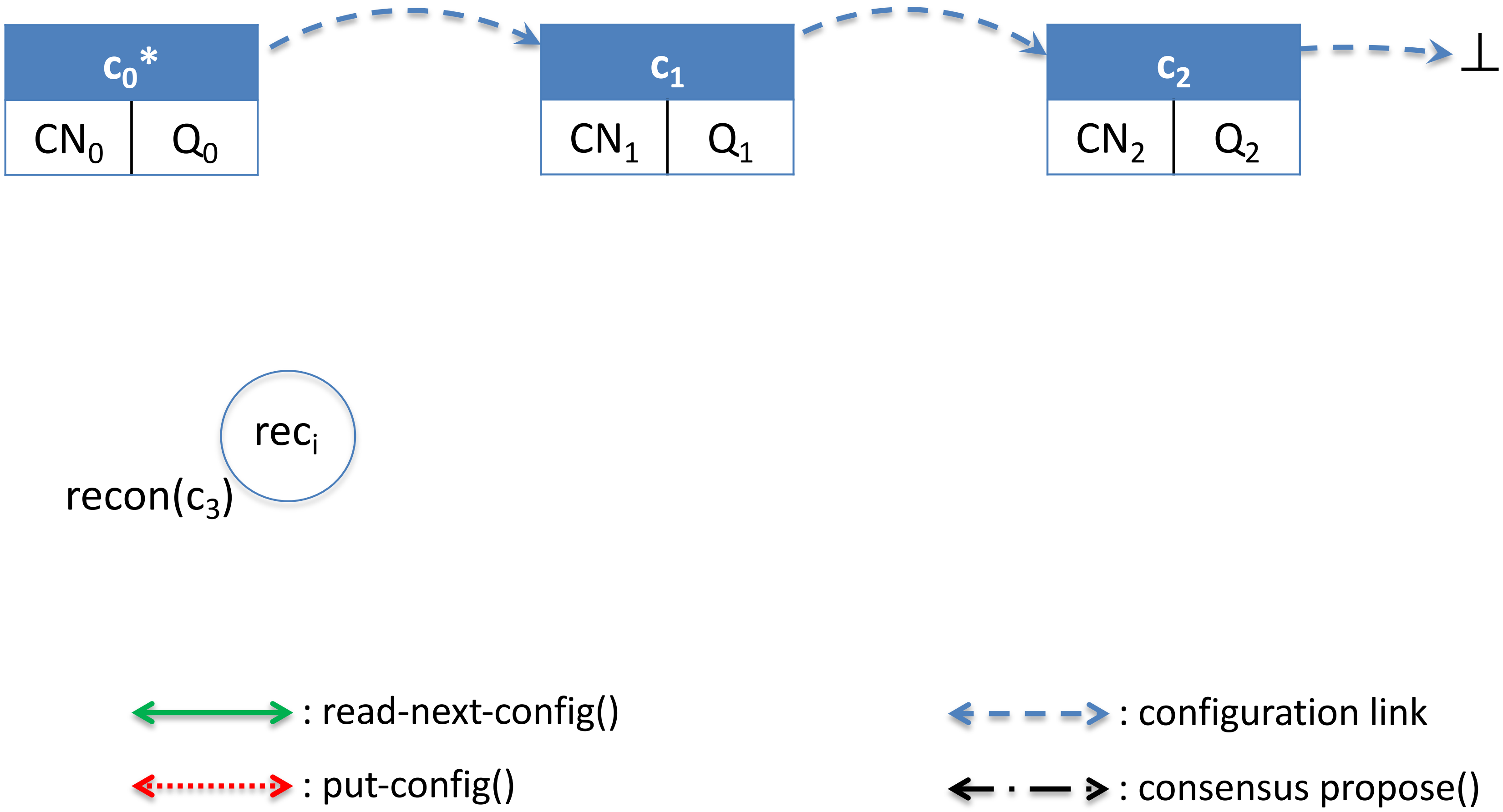
- A recon operation performs 2 major steps:
  1) Configuration *Sequence Traversal*
  2) Configuration *Installation*
     - Transfers the object state from the old to the new configuration

```
operation reconfig(c)
    if c ≠ ⊥ then
        cseq ←read-config(cseq)
        cseq ← add-config(cseq, c)
        update-config(cseq)
        cseq ← finalize-config(cseq)
end operation
```

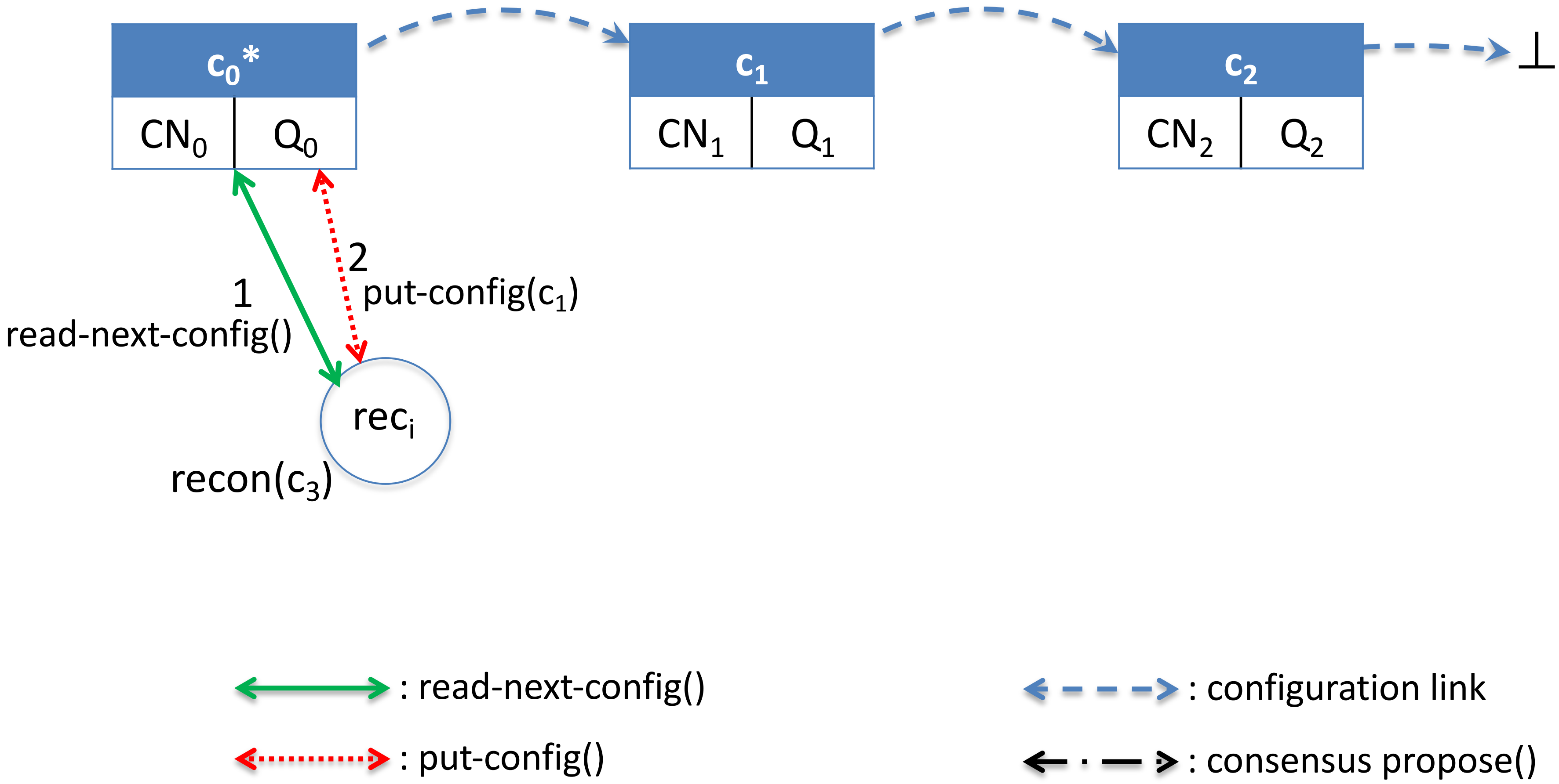
attempt get to the latest configuration } (1)

introduce the new configuration
move the latest value to the new config
let servers know it is good to be finalized } (2)

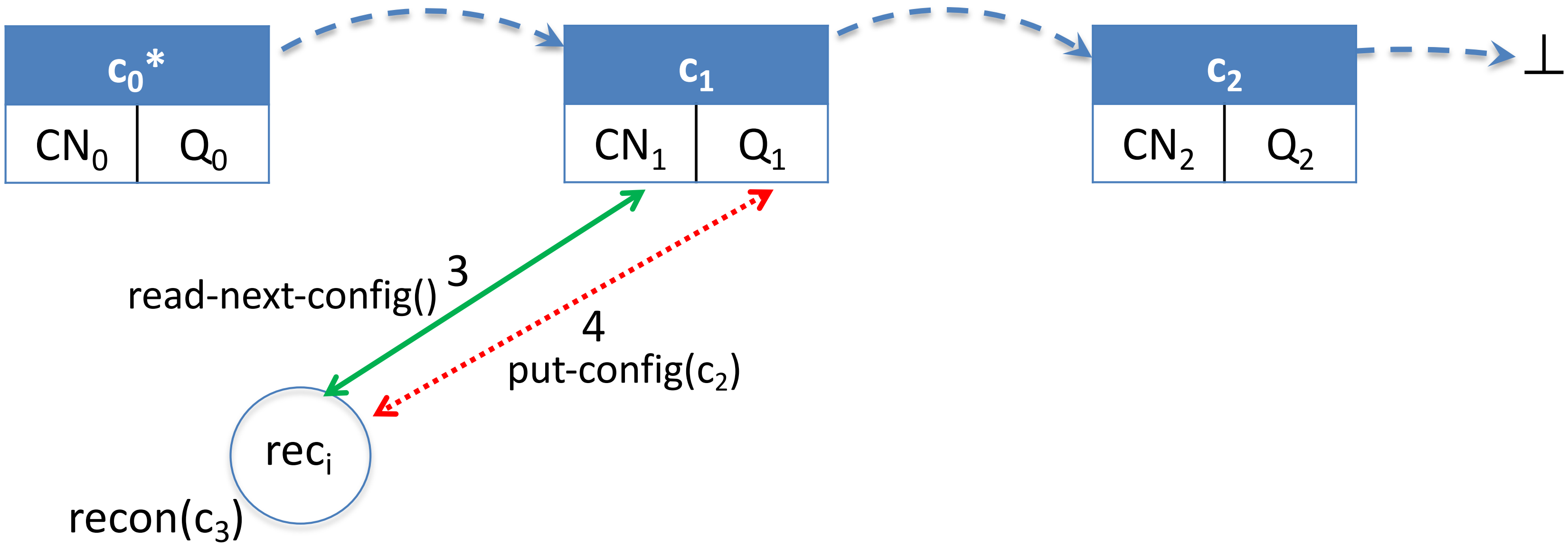
Configuration Sequence Traversal



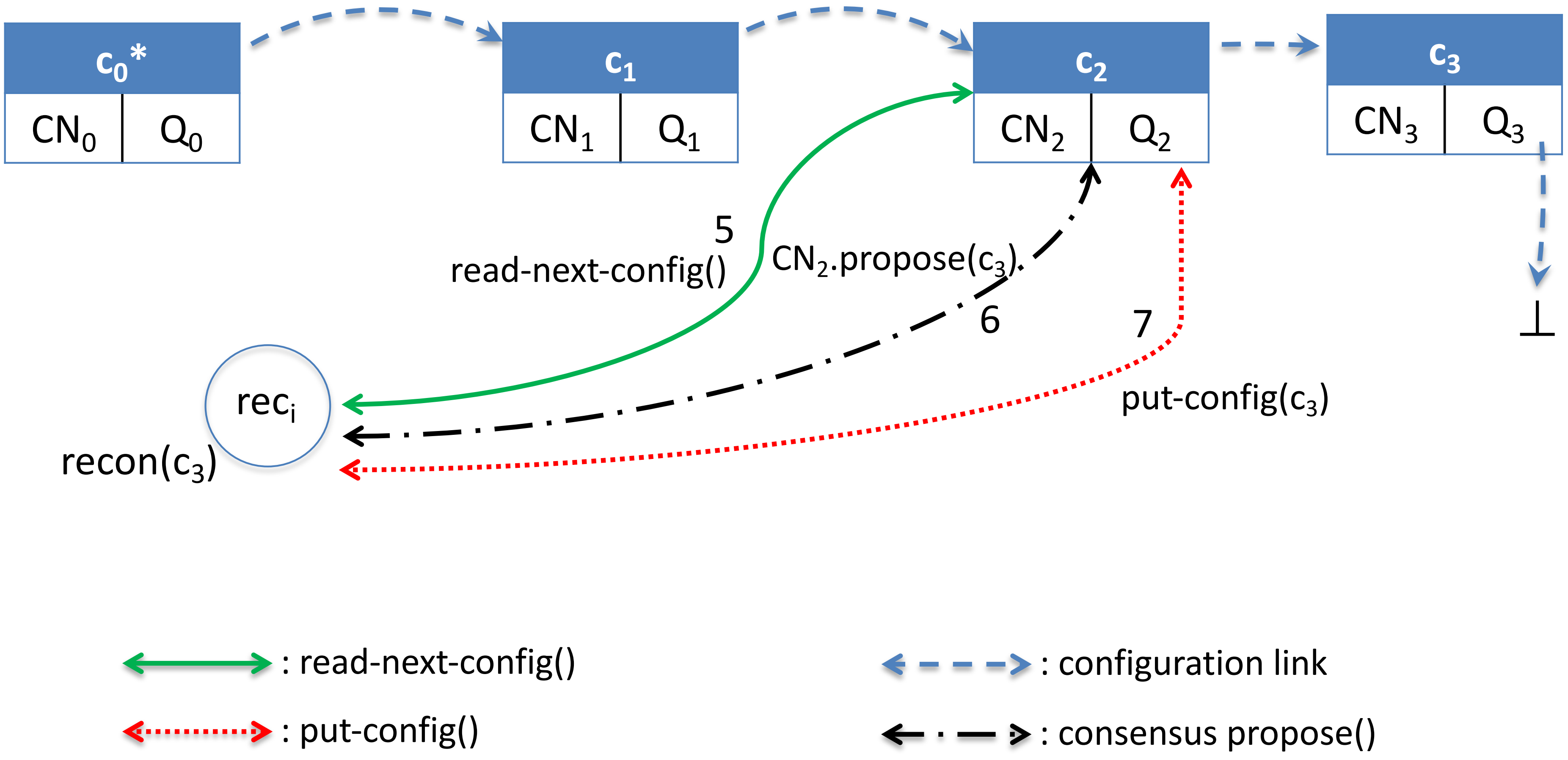
Configuration Sequence Traversal



Configuration Sequence Traversal



Configuration Sequence Traversal



Reconfiguration Service Guarantees

For any two reconfig ops π_1, π_2 s.t. π_1 before π_2

- Configuration Consistency
  - π_2 witnesses the same configuration in the i^{th} position of the sequence as π_1
- Sequence Prefix
  - the sequence witnessed by π_1 is a prefix of the sequence witnessed by π_2
- Sequence Progress
  - If the last finalized configuration witnessed by π_1 has an index i and the last finalized config witnessed by π_2 has an index j, then $i \leq j$

Reconfiguration Service Guarantees

For any two reconfig ops π_1, π_2 s.t. π_1 before π_2

- Configuration Consistency

π_1	c_0	c_1	c_2	...
π_2	c_0	c_1	c_2	...

- Sequence Prefix

π_1	c_0	c_1	
π_2	c_0	c_1	c_2

- Sequence Progress

π_1	$<c_0, F>$	$<c_1, P>$	$<c_2, P>$	
π_2	$<c_0, F>$	$<c_1, P>$	$<c_{2,}, F>$	...

Read/Write Operations using DAPs

Reader Protocol

- Traverse Config Sequence cseq
- Find μ = max(<c, F>) in cseq
- Set v = last(<c,*>) in cseq
- Discover for μ ≤ i ≤ v
  (t,v)=max(cseq[i].**get-data**())
- Do
  - cseq[v].**put-data**(t,v)
  - Traverse Sequence cseq
- while(|cseq| > v)

Writer Protocol(val) (at w_i)

- Traverse Config Sequence cseq
- Find μ = max(<c, F>) in cseq
- Set v = last(<c,*>) in cseq
- Discover for μ ≤ i ≤ v
  t_{max}=max(cseq[i].**get-tag**())
- (t,v)= ($\langle t_{max}+1, w_i \rangle$, val)
- Do
  - cseq[v].**put-data**(t,v)
  - Traverse Sequence cseq
- while(|cseq| > v)

DAP Implementation using EC

- Servers maintain a List of the last δ coded elements they received
- Processes requests:
  - Get-tag(): Max tag from the list of servers
  - Get-data(): get list of servers to try to decode some value
  - Put-data(): send tag t and coded element e_j to server s_j

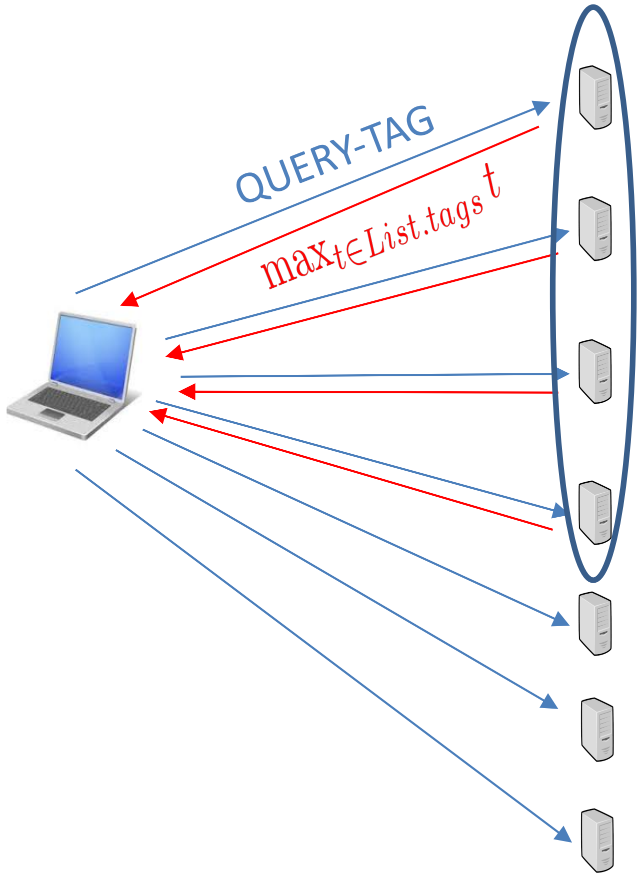
Get-tag()

c.get-tag() at p_i

- Request tag from n+k/2 servers in c.Servers
- Discover t_{max}=max(t) from the received replies
- Return t_{max}

Receive(Query-Tag) at server s_j

- Find t_{max} within $List_j$
- Send t_{max} to the requester



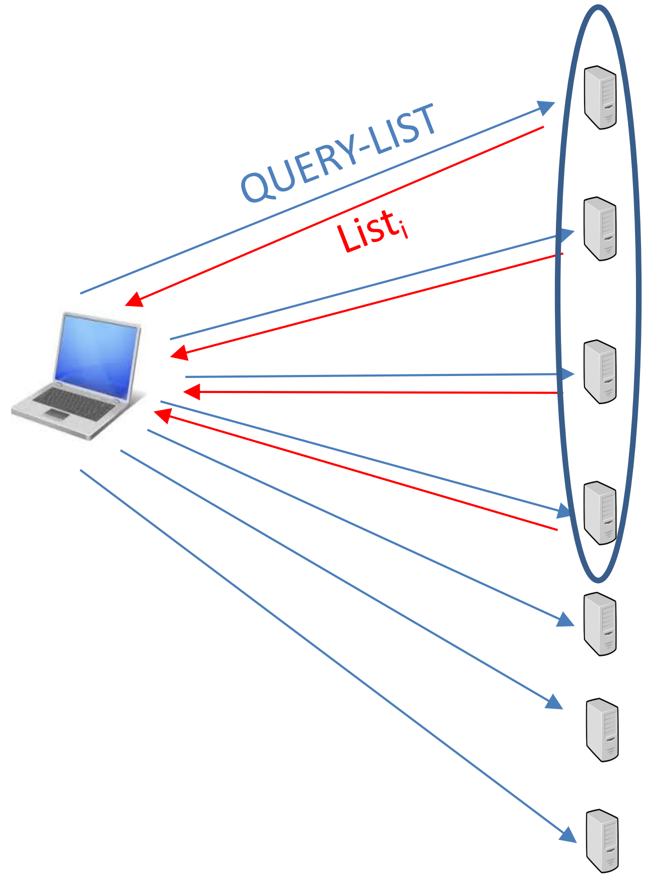
Get-data()

c.get-data() at p_i

- Request List from n+k/2 servers in c.Servers
- Discover t_{max} from the received Lists s.t. its value v is decodable
- Return (t_{max}, v)

Receive(Query-List) at server s_j

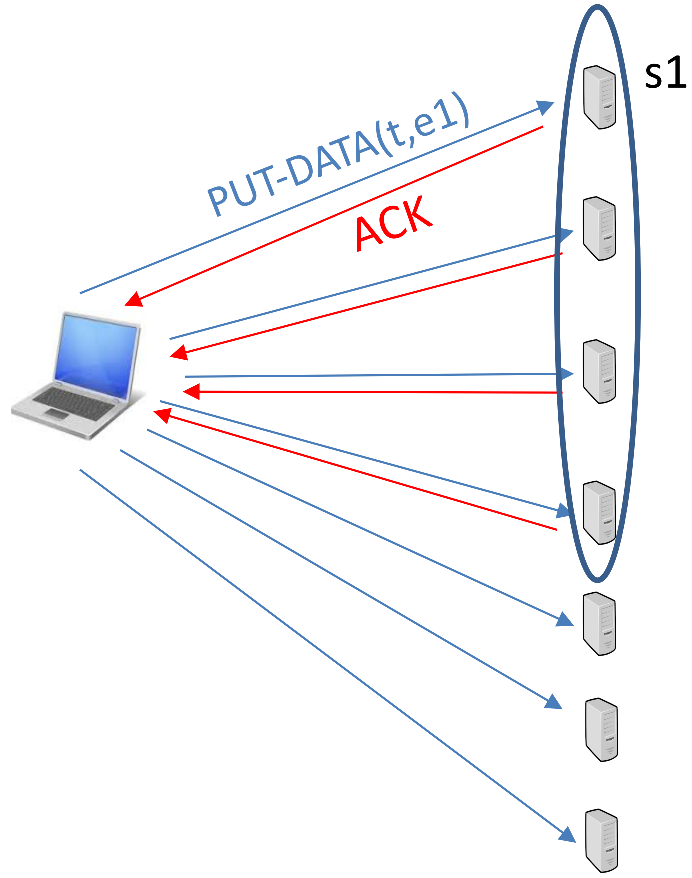
- Reply with $List_j$



Put-data(<t,v>)

c.put-data(<t,v>) at p_i

- Generate code elements [(t,e_1), …, (t,e_n)] where $e_i = \Phi_i(v)$
- Send (t,e_i) to $s_i \in$ c.Servers
- Wait until receiving ACK from n+k/2 servers in c.Servers



Receive(Put-Data, <t,e_j>) at server s_j

- Add <t,e_j> in $List_j$
- If $|List_j| > \delta+1$
  - Find t_{min} in $List_j$ and remove any <t_{min}, *> pairs from $List_j$
  - Add element <t_{min},⊥ > in $List_j$
- Reply with ACK

Properties of DAP implementation

- MDS code Based Algorithm
- Uses [n, k] MDS codes, *n vs n/k*
- Any client may crash fail and at most servers can experience crash failure
- Always safe
- If the number of write operations concurrent with a read operation is upper bounded by then the read and write operations are live
- First two-round erasure-code-based atomic memory algorithm

Modular & Adaptive Implementation

- Different DAPs per Configuration

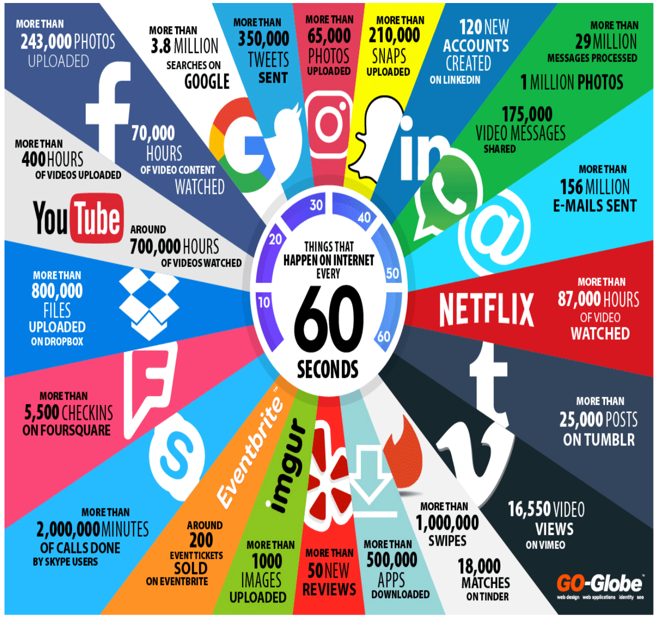
- Regardless their implementation the DAPs
  - Serve the same purpose in any configuration
    - Get-tag: returns the max tag in the configuration
    - Get-data: returns data associated with a tag
    - Put-data: alters the data associated with a tag
  - Can yield atomic implementations if they satisfy properties C1 and C2

Conclusions

- We presented ARES
  - Reconfigurable
  - Atomic Read/Write Operations
  - Use of DAPs for
    - Modularity: Reads/writes omniscient of the underly DAP implementation
    - Adaptiveness: usage of any algorithm per configuration
  - First implementation of Erasure Coded read/write operations in a reconfigurable setting

Thank You!

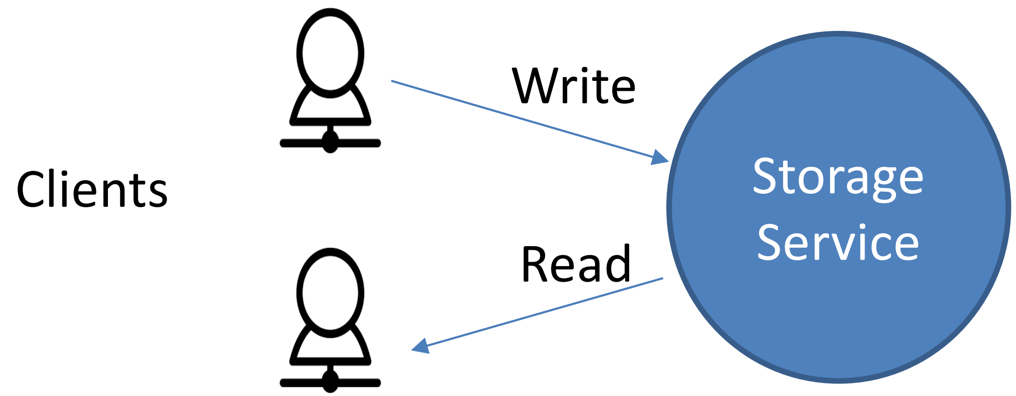
What is Common Among These Applications?



All these Applications Use Storage as Service at its Core

Storage Systems providing “suitable guarantees” are essential for application design

Consistency Guarantees of a Storage Service



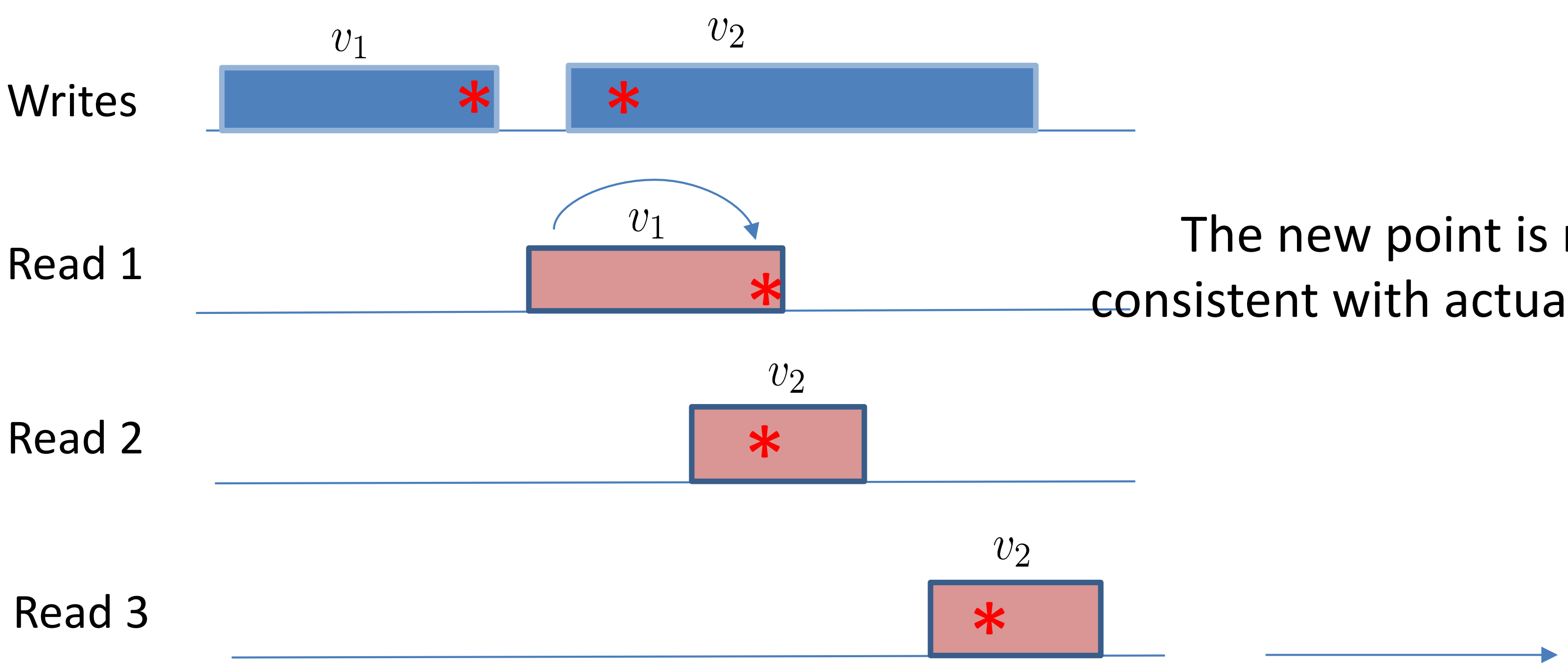
Consistency Guarantee	Quick Definition	Application View Point	Storage Service Design
Strong Consistency	Read returns last completed Write	Preferred	Costly, Complex Algorithms
Weak Consistency (e.g. Eventual Consistency)	Read eventually returns a completed write	Not Preferred, behavior different from a single threaded program	Relatively less costly, easier algorithms

Consistency in Various Storage Systems

System	Consistency Notion
Facebook TAO	Eventual Consistency
Amazon Dynamo	Eventual Consistency
OpenStack Swift	Eventual Consistency
Cassandra	Eventual Consistency/Strong Consistency
Microsoft Azure Store	Strong Consistency

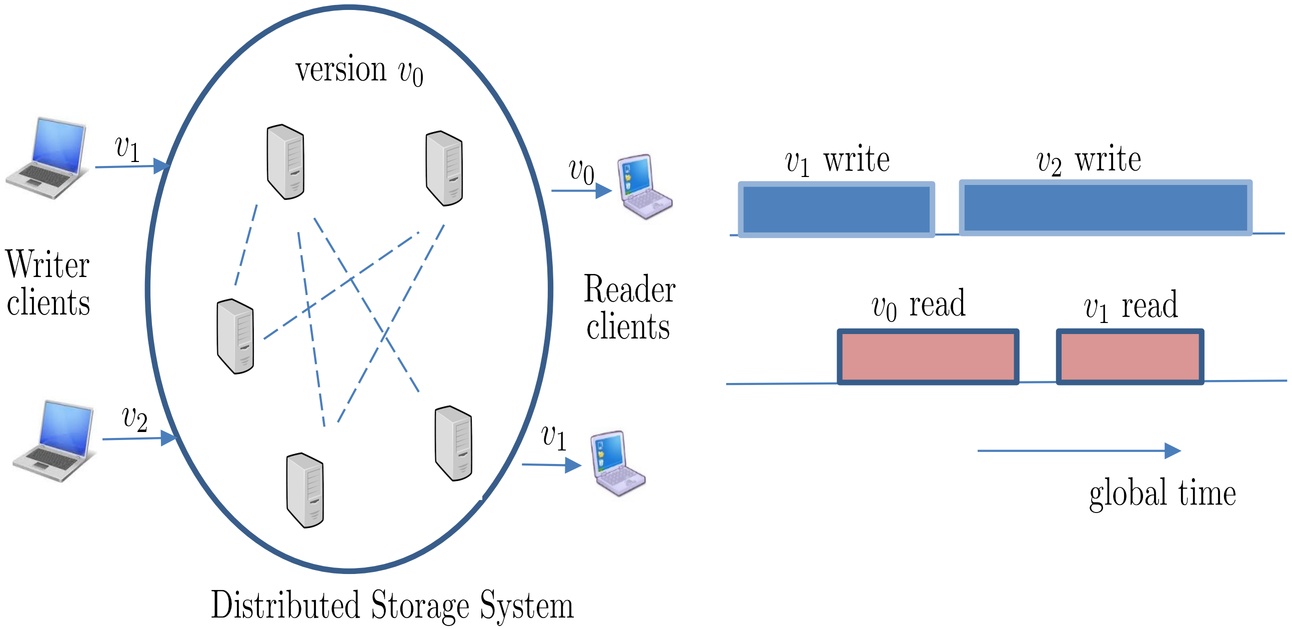
Atomicity

- Shrink the duration of each operation to a chosen serialization point between the operation's invocation and response, such that
- the external behavior of reads/writes is consistent with the the ordering of the serialization points.

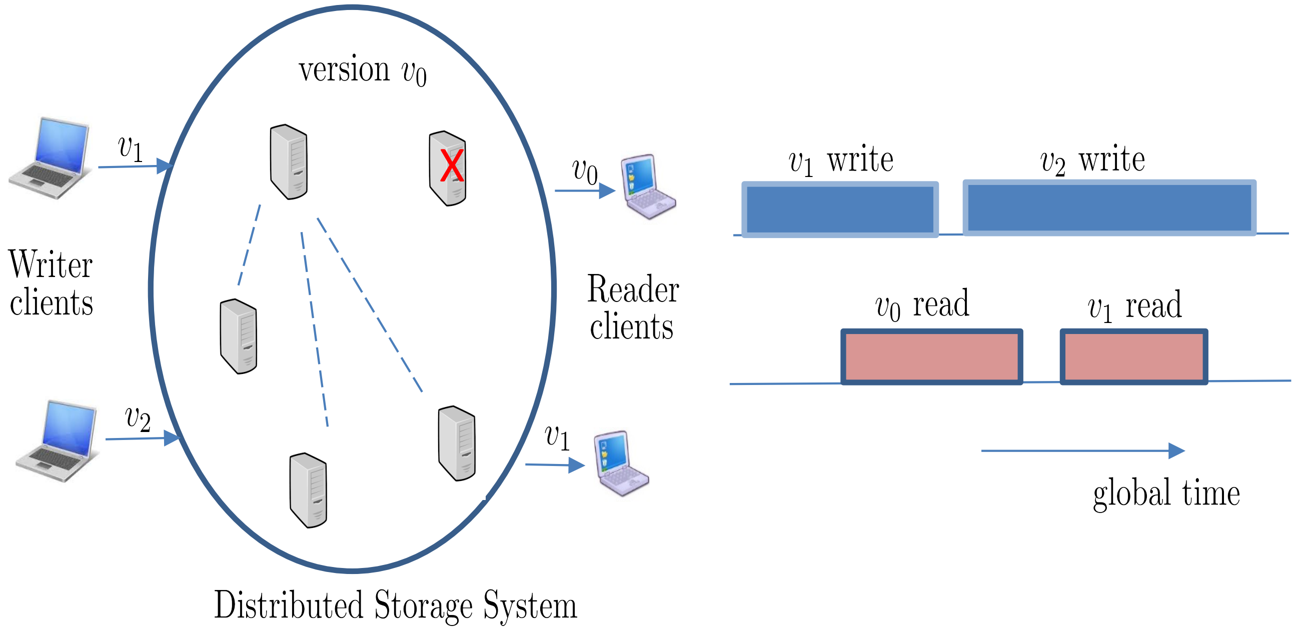


The new point is not consistent with actual events

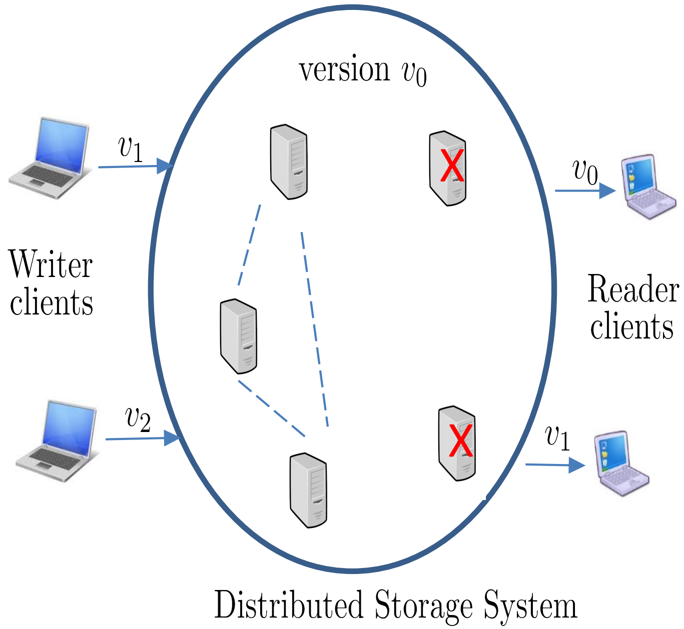
Reconfigurable Distributed Storage System

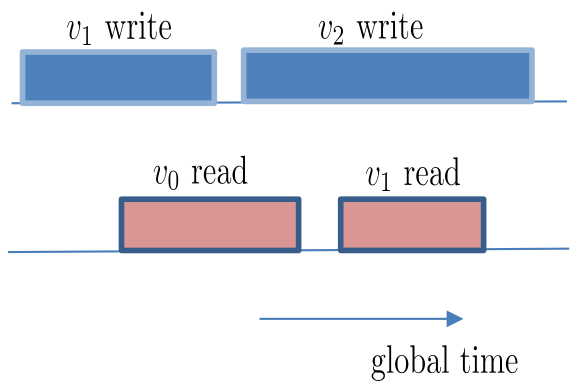


Reconfigurable Distributed Storage System (cont'd)

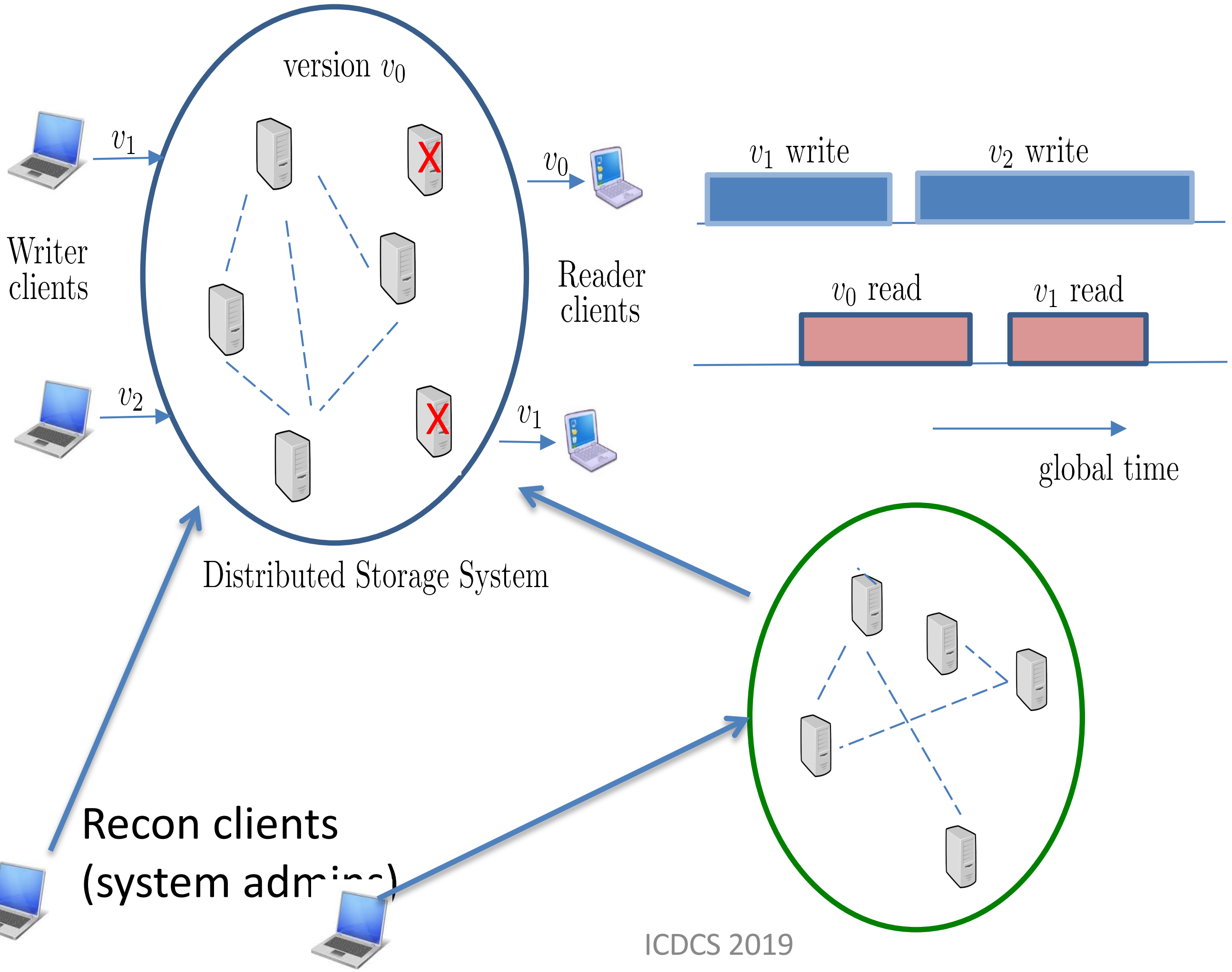


Reconfigurable Distributed Storage System (cont'd)

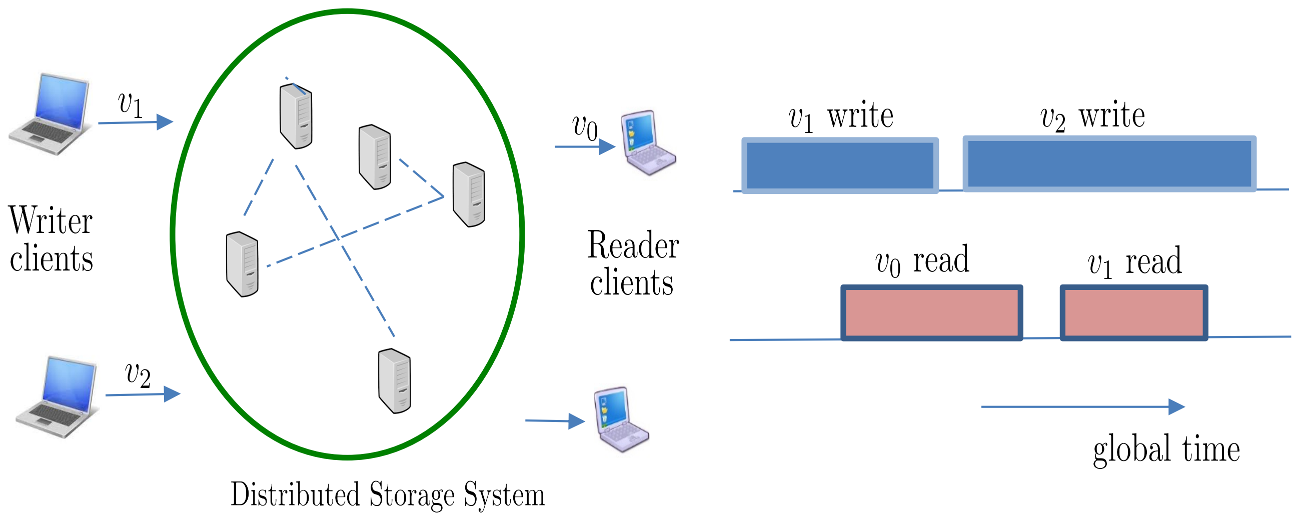




Reconfigurable Distributed Storage System (cont'd)



Reconfigurable Distributed Storage System (cont'd)



Problems due to lack of consistency (scenario 1)

- Alice: I lost my wedding ring.
- Alice: Thank god! I found it!
- Bob: I am glad to hear that!

Bob

- Alice: I lost my wedding ring.

  Alice: Thank god! I found it!

  Bob: I am glad to hear that!

Alice

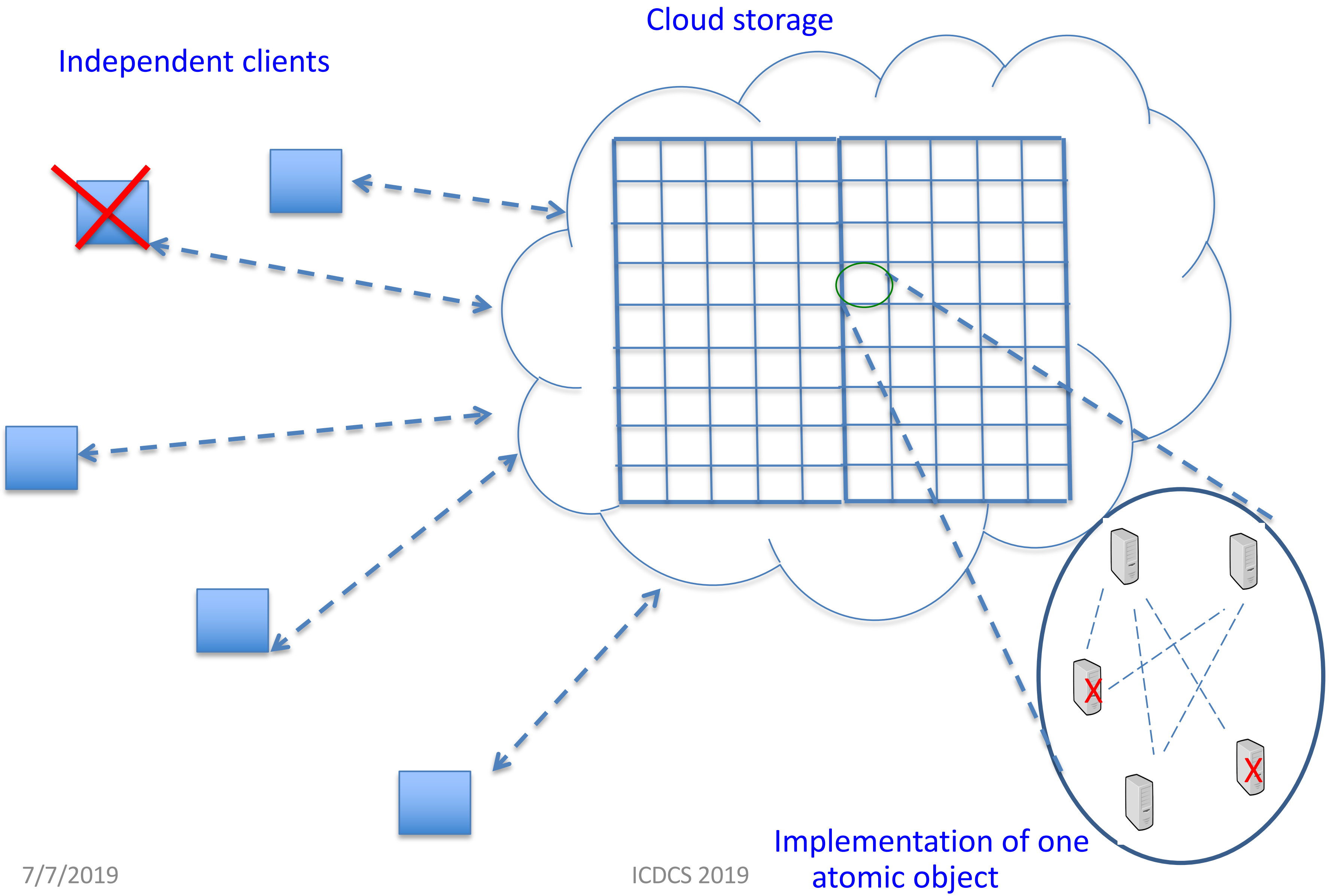
Rachel

- Alice: I lost my wedding ring.
- Bob: I am glad to hear that!

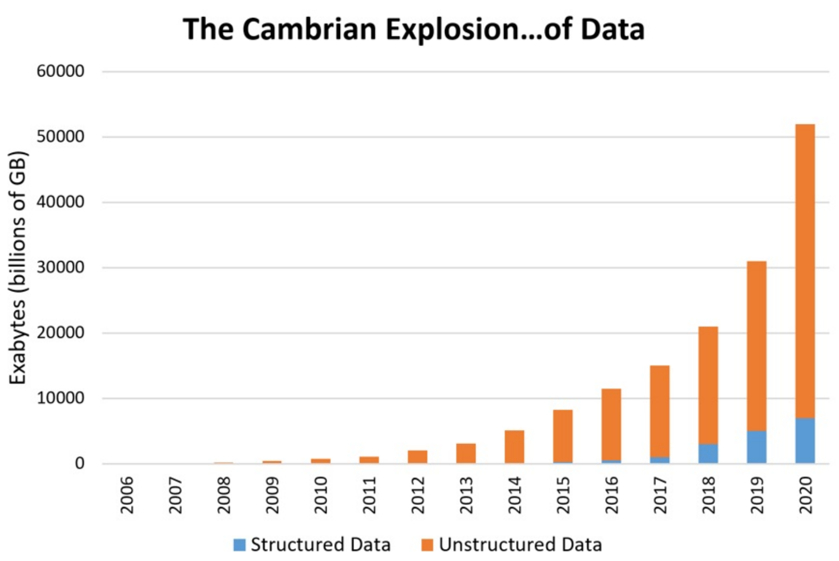
…….

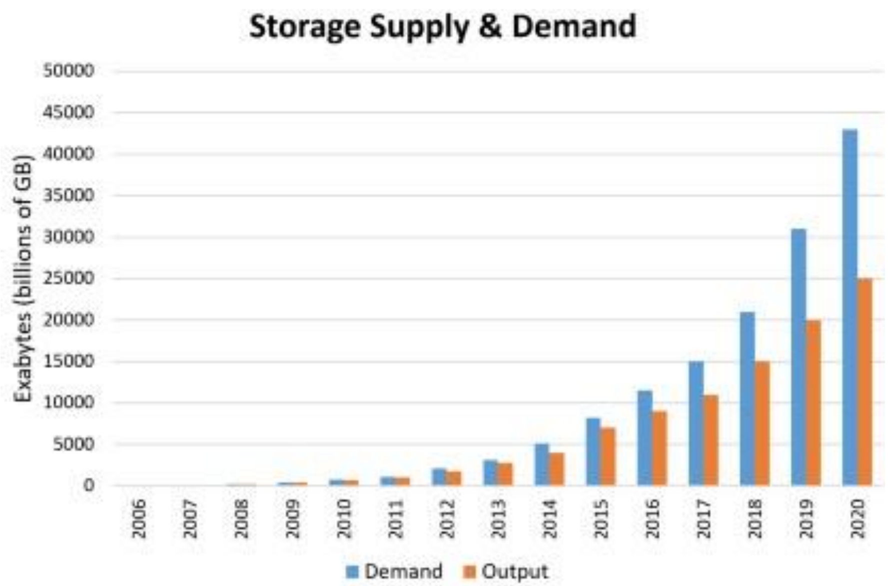
- Alice: Thank god! I found it!

Atomicity and Shared Memory



Should we bother about Storage Cost? (A big Yes !)





Source: EEtimes Article, https://www.eetimes.com/author.asp?section_id=36&doc_id=1330462

Object Storage is one of the main techniques to handle Unstructured Data

Who uses Erasure codes for Storage ?

System	Code
Google File System	MDS Code
(Facebook) HDFS-RAID (Back-Up)	MDS Code
Microsoft Azure/Giza (Strongly Consistent, Consensus based)	Local Reconstruction Codes + MDS Codes

- Erasure Codes have been traditionally used for efficient storage of Write-Once Data
- Recent Works Show benefits of Erasure Codes for Consistent Data Storage as well

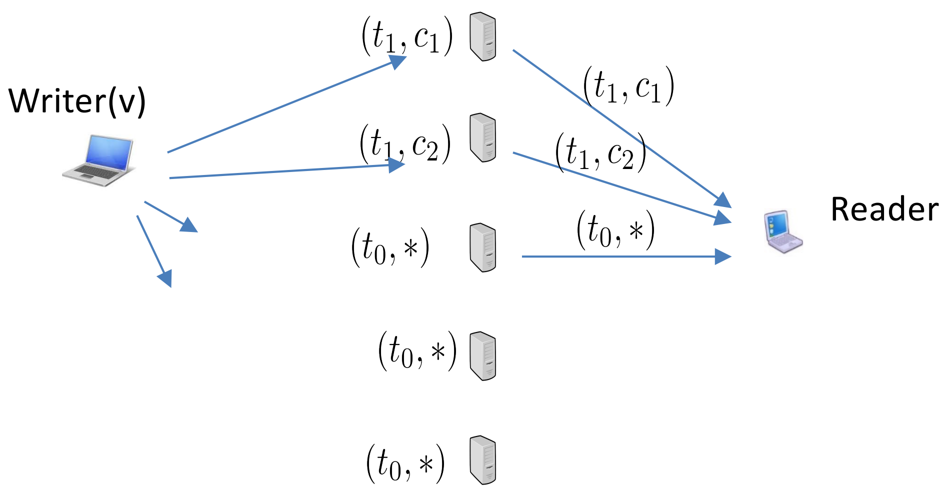
Efficient Erasure codes for Data Storage is an Active Area of Research

Code	Main Use	Where is it Used?
Local Reconstruction Codes	Fast Degraded Reads	Microsoft Azure
Regenerating Codes	Low Bandwidth Repair of Crashed Servers	Networked Storage Systems
Random Linear Network Codes (RLNC)	Ideal for Decentralized Operation	Peer-to-Peer Systems, Edge Caching (Ask Vitaly!), etc
Codes for Clustered Systems (hybrid codes)	Flexible trade-off of intra vs inter cluster bandwidth costs	Geo distributed Data Centers

All the above codes have significantly lower storage overhead than replication for the same fault tolerance

We can build algorithms on top of any of these coded storage systems and still guarantee consistency properties

Specific Challenge while Using Erasure Codes for Consistent Storage : Concurrent Writes



- Write Concurrent with Read
- Reader potentially gets coded values corresponding to different tags

The main Algorithmic Challenge is Ensuring Liveness of Read Operations (Decodability) in the presence of Concurrent Writes

Erasure Code –Based Leaderless Algorithms for Strong Consistency (Our works)

1. The SODA Algorithm (IEEE IPDPS 2016)
   - Optimizes Storage Cost at the Expense of Write Cost
2. The RADON Repair (OPODIS 2016)
   - Permits Online Repair of Crashed Servers
3. The Layered Data Storage (LDS) Algorithm (ACM PODC 2017)
   - Modularizes Implementations of Consistency and Erasure Codes

Properties of TREAS

- MDS code Based Algorithm
- Uses [n, k] MDS codes, *n vs n/k*
- Any client may crash fail and at most $\frac{n-k}{2}$ servers can experience crash failure
- Always safe
- If the number of write operations concurrent with a read operation is upper bounded by δ then the read and write operations are live
- First two-round erasure-code-based atomic memory algorithm

Storage and Communication Costs

Algorithm	Storage Cost	Read Communication Cost	Write Communication Cost
ABD	*n*	*2n*	*n*
TREAS	$\frac{n}{k}(\delta + 1)$	$\frac{n}{k}(\delta + 1)$	$\frac{n}{k}$

e.g., number of servers n = 20 and [n k] MDS code with k = 10

Atomicity in terms of DAP

operation read()
  $\langle t, v\rangle \leftarrow c$.get-data()
  c.put-data($\langle t, v\rangle$)
  return $\langle t, v\rangle$
end operation

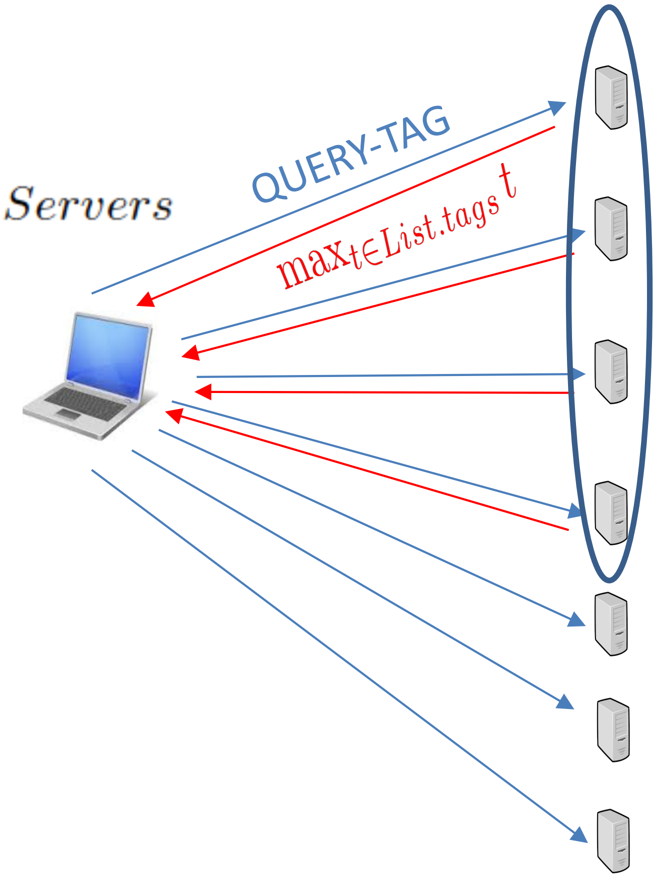
operation write(v)
  $t \leftarrow c$.get-tag()
  $t_w \leftarrow \langle t.z + 1, w\rangle$
  c.put-data($\langle t_w, v\rangle$)
end operation

Suppose the DAP implementation satisfies the consistency properties C1 and C2. Then any execution the above protocol in a configuration configuration, is atomic and liveness of the algorithms is possible if DAPs are live

Get-tag()

At a process p_i:

procedure c.get-tag()
  send (QUERY-TAG) to each $s \in c.Servers$
  until p_i receives $\langle t_s, e_s \rangle$ from $\lceil \frac{n+k}{2} \rceil$ servers in $c.Servers$
  $t_{max} \leftarrow \max(\{t_s : \text{ received } \langle t_s, v_s \rangle \text{ from } s\})$
  return t_{max}
end procedure



At a server s_j:

Upon receive (QUERY-TAG) $_{s_i, c_k}$ **from** q
  $\tau_{max} = \max_{(t,c) \in List} t$
  Send τ_{max} to q
end receive

Read Operation : get data

$\perp$

$L_1 = \{ (t_{20}, c_{9,1}),$ $, (t_{17}, c_{17,1}),$ $(t_{15}, c_{15,1}), (t_{14}, \perp),$......................}

$L_2 = \{$ $(t_{16}, c_{16,4}), (t_{15}, c_{15,4}), (t_{14}, c_{14,2})$........................}

$L_3 = \{$ $(t_{18}, c_{18,3}),$ $(t_{16}, c_{16,3}),$ $,$ $(t_{14}, c_{14,3})$......................}

$L_4 = \{ (t_{20}, \perp), (t_{19}, c_{19,4}), (t_{18}, \perp),$ $(t_{16}, c_{16,4}), (t_{15}, c_{15,4}),$ ....}

$L_5 = \{$ $(t_{18}, c_{18,5}),$ $, (t_{15}, c_{15,5}),$ ....$(t_{13}, c_{13,5})$..}

$L_6 = \{$ $(t_{18}, c_{18,6}),$ $(t_{16}, c_{16,6}),$ $, (t_{14}, c_{14,6})$......................}

$L_7 = \{$ $, (t_{19}, \perp),$ $(t_{18}, c_{18,7}),$ $(t_{16}, c_{16,7}), (t_{15}, c_{15,7}),$ .... }

decode $[n = 7, k = 3]$

(t_{18}, v_{18})

TREAS: implementation

read

at each process $p_i \in \mathcal{I}$**:**
  operation read()
    $\langle t, v \rangle \leftarrow c$.get-data()
    c.put-data($\langle t, v \rangle$)
    return $\langle t, v \rangle$
  end operation

  operation write(v)
    $t \leftarrow c$.get-tag()
    $t_w \leftarrow \langle t.z + 1, w \rangle$
    c.put-data($\langle t_w, v \rangle$)
  end operation

write

  procedure c.get-tag()
    send (QUERY-TAG) to each $s \in c.servers$
    until p_i receives $\langle t_s, e_s \rangle$ from $\left\lceil \frac{n+k}{2} \right\rceil$ servers in $c.servers$
    $t_{max} \leftarrow \max(\{t_s :$ received $\langle t_s, v_s \rangle$ from $s\})$
    return t_{max}
  end procedure

TREAS: implementation (cont'd)

procedure c.get-data()
  send (QUERY-LIST) to each $s \in c.servers$
  until p_i receives $List_s$ from each server $s \in \mathcal{S}_g$ s.t. $|\mathcal{S}_g| = \left\lceil \frac{n+k}{2} \right\rceil$ and $\mathcal{S}_g \subset c.servers$
  $Tags_*^{\geq k}$ = set of tags that appears in k lists
  $Tags_c^{\geq k}$ = set of tags that appears in k lists with values
  $t_{max}^* \leftarrow Tags_*^{\geq k}$
  $t_{max}^{dec} \leftarrow Tags_c^{\geq k}$
  if $t_{max}^c = t_{max}^*$ **then**
    $v \leftarrow$ decode from t_{max}^{dec}
  return $\langle t_{max}^{dec}, v \rangle$
end procedure

procedure c.put-data($\langle \tau, v \rangle$))
  $code\text{-}elems = [(\tau, e_1), \ldots, (\tau, e_n)]$, $e_i = \Phi_i(v)$
  send (WRITE, $\langle \tau, e_i \rangle$) to each $s_i \in c.servers$
  until p_i receives ACK from $\left\lceil \frac{n+k}{2} \right\rceil$ servers in $c.servers$
end procedure

TREAS: implementation (cont'd)

At a server

at each server $s_i \in \mathcal{S}$ **in configuration** c_k**:**
State Variables:
$List \subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\{(t_0, \Phi_i(v_0))\}$

Upon receive (QUERY-TAG) $_{s_i, c_k}$ **from** q
  $\tau_{max} = \max_{(t,c) \in List} t$
  Send τ_{max} to q
end receive

Upon receive (QUERY-LIST) $_{s_i, c_k}$ **from** q
  Send $List$ to q
end receive

Upon receive (PUT-DATA, $\langle \tau, e_i \rangle$) $_{s_i, c_k}$ **from** q
  $List \leftarrow List \cup \{\langle \tau, e_i \rangle\}$
  if $|List| > \delta + 1$ **then**
    $\tau_{min} \leftarrow \min\{t : \langle t, * \rangle \in List\}$
  // remove the coded value and retain the tag
    $List \leftarrow List \backslash \{\langle \tau, e \rangle : \tau = \tau_{min} \wedge \langle \tau, e \rangle \in List\} \cup \{(\tau_{min}, \bot)\}$
end receive

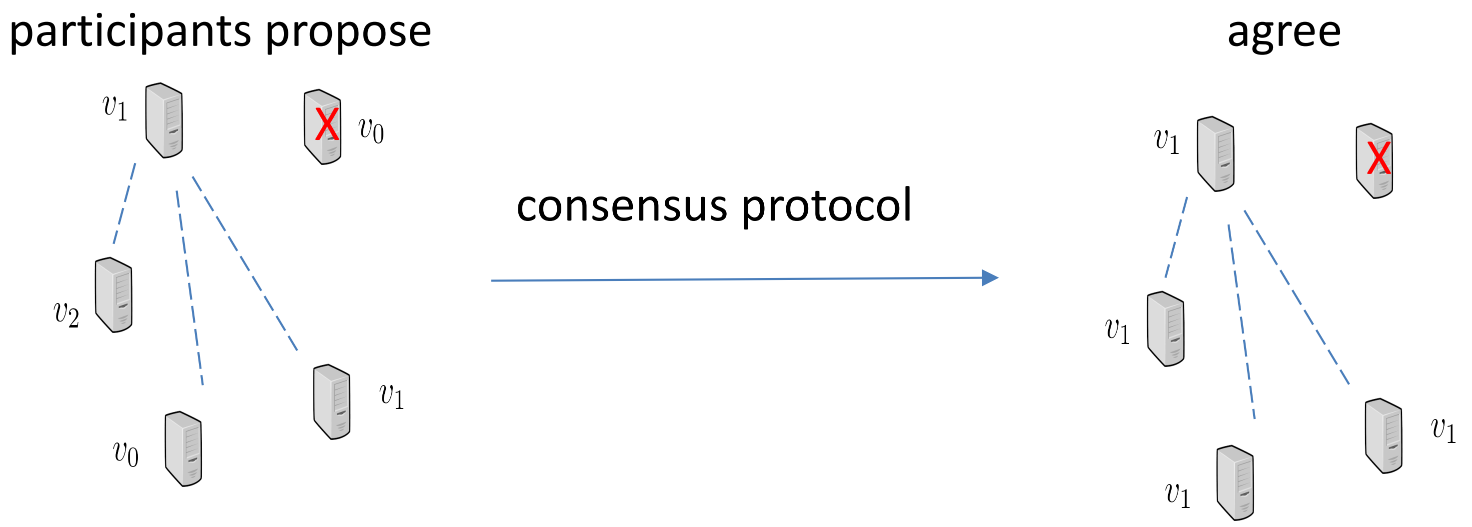
Storage and Communication Costs

Algorithm	Storage Cost	Read Communication Cost	Write Communication Cost
ABD	*n*	*2n*	*n*
TREAS	$\frac{n}{k}(\delta + 1)$	$\frac{n}{k}(\delta + 1)$	$\frac{n}{k}$

e.g., number of servers n = 20 and [n k] MDS code with k = 10

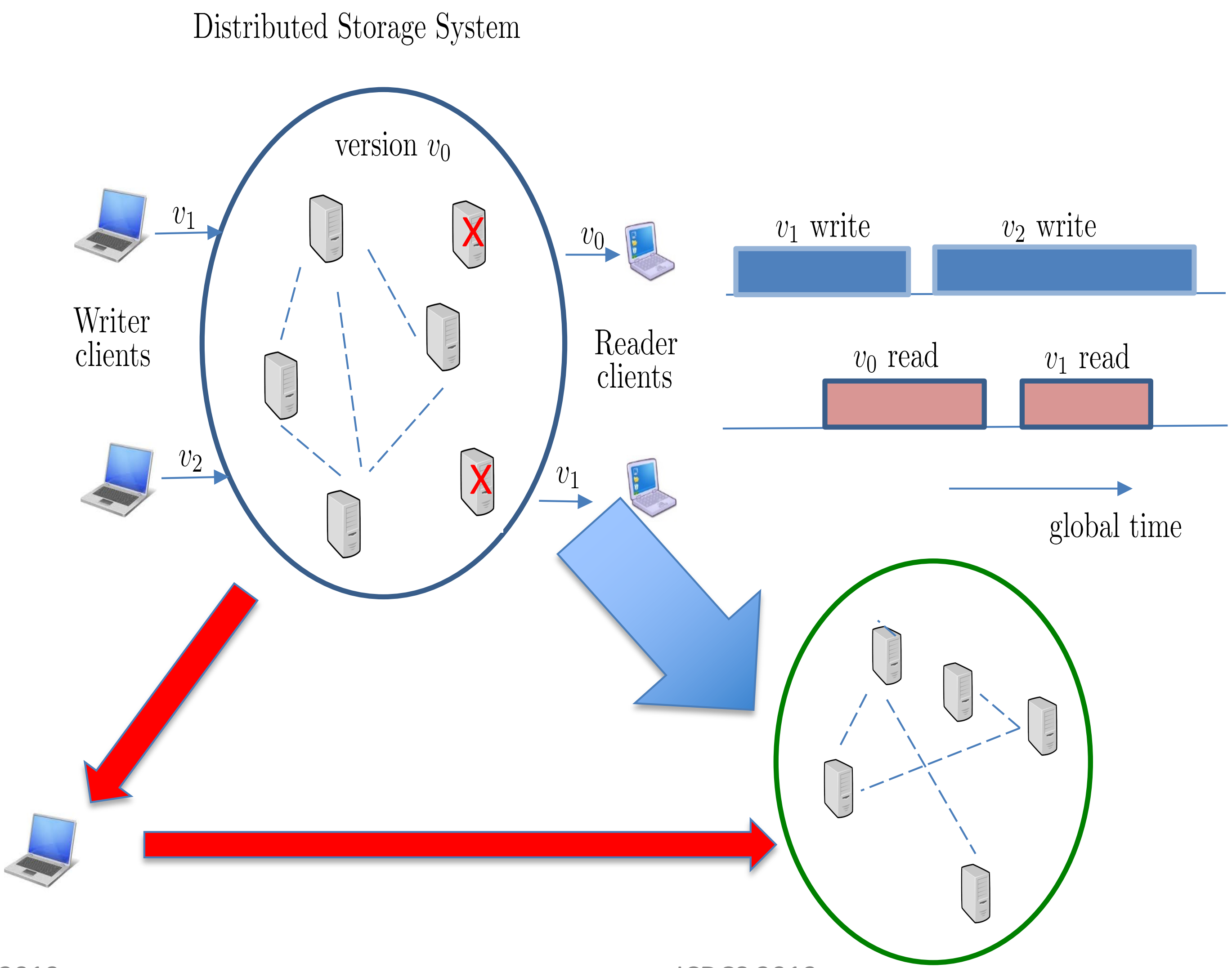
Configuration

- A set of servers, each with an unique id
- The server side responses of the algorithm
- An instance of the consensus service running on top of the set of servers in the configuration



- FLP not live
- Cryptocurrency byzantine setting

Moving data during reconfiguration



Recon client

RECS: reconfiguration operation

at each reconfigurer rec_i

State Variables:

$cseq[] s.t. cseq[j] \in \mathcal{C} \times \{F, P\}$ with members:

Initialization:

$cseq[0] = \langle c_0, F \rangle$

operation reconfig(c)

  if $c \neq \bot$ **then**

    $cseq \leftarrow$ read-config$(cseq)$

    $cseq \leftarrow$ add-config$(cseq, c)$

    update-config$(cseq)$

    $cseq \leftarrow$ finalize-config$(cseq)$

end operation

procedure update-config(seq)

  $\mu \leftarrow \max(\{j : seq[j].status = F\})$

  $\nu \leftarrow |seq|$

  $M \leftarrow \emptyset$

  for $i = \mu : \nu$ **do**

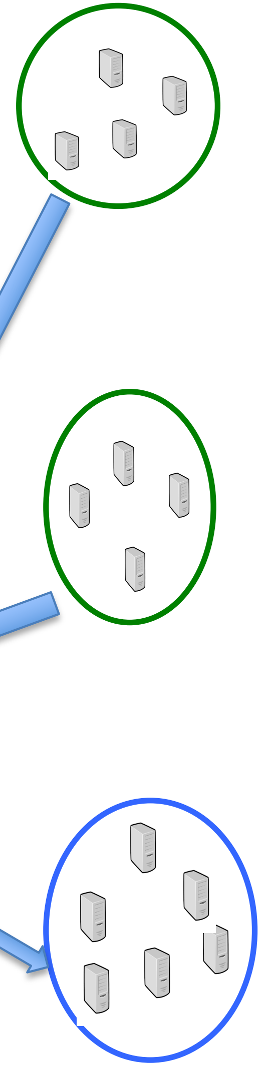
    $\langle t, v \rangle \leftarrow$ get-data$(seq[i].cfg)$

    $M \leftarrow M \cup \{\langle \tau, v \rangle\}$

  $\langle \tau, v \rangle \leftarrow \max_t\{\langle t, v \rangle : \langle t, v \rangle \in M\}$

  put-data$(seq[\nu], \langle \tau, v \rangle)$

end procedure

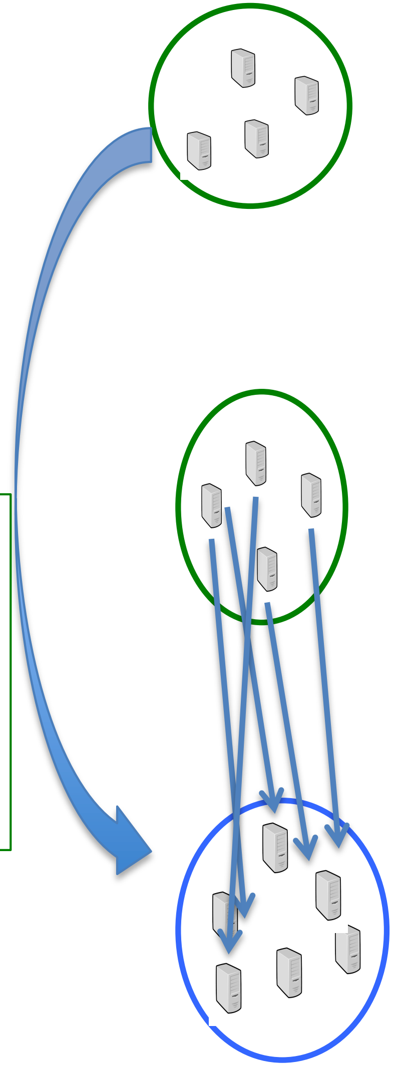


RECS: reconfiguration operation

```
at each reconfigurer rec_i
State Variables:
    cseq[] s.t. cseq[j] ∈ C × {F, P} with members:
Initialization:
    cseq[0] = ⟨c_0, F⟩

operation reconfig(c)
    if c ≠ ⊥ then
        cseq ← read-config(cseq)
        cseq ← add-config(cseq, c)
        update-config(cseq)
        cseq ← finalize-config(cseq)
end operation
```

```
procedure update-config(seq)
    μ ← max({j : seq[j].status = F})
    ν ← |seq|
    M ← ∅
    for i = μ : ν do
        ⟨t, v⟩ ← get-data(seq[i].cfg)
        M ← M ∪ {⟨τ, v⟩}
    ⟨τ, v⟩ ← max_t{⟨t, v⟩ : ⟨t, v⟩ ∈ M}
    put-data(seq[ν], ⟨τ, v⟩)
end procedure
```



Performance of RECS

- RECS: Always atomic
- In the absence of reconfigurations: liveness of the read/write operations is dependent on the liveness of the DAP primitives

- In case of reconfigurations:
  - messages arrive within the time interval [d, D]
  - k is the number of reconfigurations in the entire execution or within a sufficiently long interval
  - k is fixed then d can be arbitrarily small – liveness
  - Reconfigurations are infinitely often, without any bound on d --- cannot guarantee liveness
  - Reconfigurations are infinitely many, there exists a minimum bound on d --- can guarantee liveness