NETYS 2017, Marrakech, Morocco

# Atomic Shared Objects for Distributed Systems: Consistency, Latency, Reconfiguration

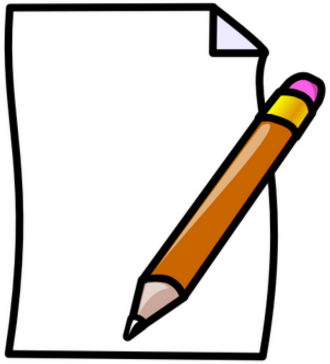

Alexander A. Schwarzmann
University of Connecticut, USA

# قراءة , كتابة , الحساب

- **“Three R’s” -- Reading**, w’**Riting**, and a’**Rithmetic**
  - Underlay much of human intellectual activity
  - Venerable foundation of computing technology
- With networking, *communication* became a major activity
  - Email – electronic counterpart of postal service
- Yet, it is natural to deal with *reading*, *writing*, and *computing*
  - A web browser app may *load* (i.e., *read*) a page, perform computation, and *save* (i.e., *write*) the results
  - In distributed databases we *retrieve* and *store* data, and rarely talk about sending and receiving data
- Arguably, it is also easier to develop distributed algorithms with readable/writeable objects, than to use message passing

# Sharing Memory in a Networked System

- Let's place a shareable object at a node in a network
  - Not fault-tolerant – single point of failure
  - Not efficient – performance bottleneck
  - Not very available, does not provide longevity, etc…



# Sharing Memory in a Networked System

- So we replicate – we'd have to anyway, since redundancy is the only means for providing fault-tolerance

# Sharing Memory in a Networked System

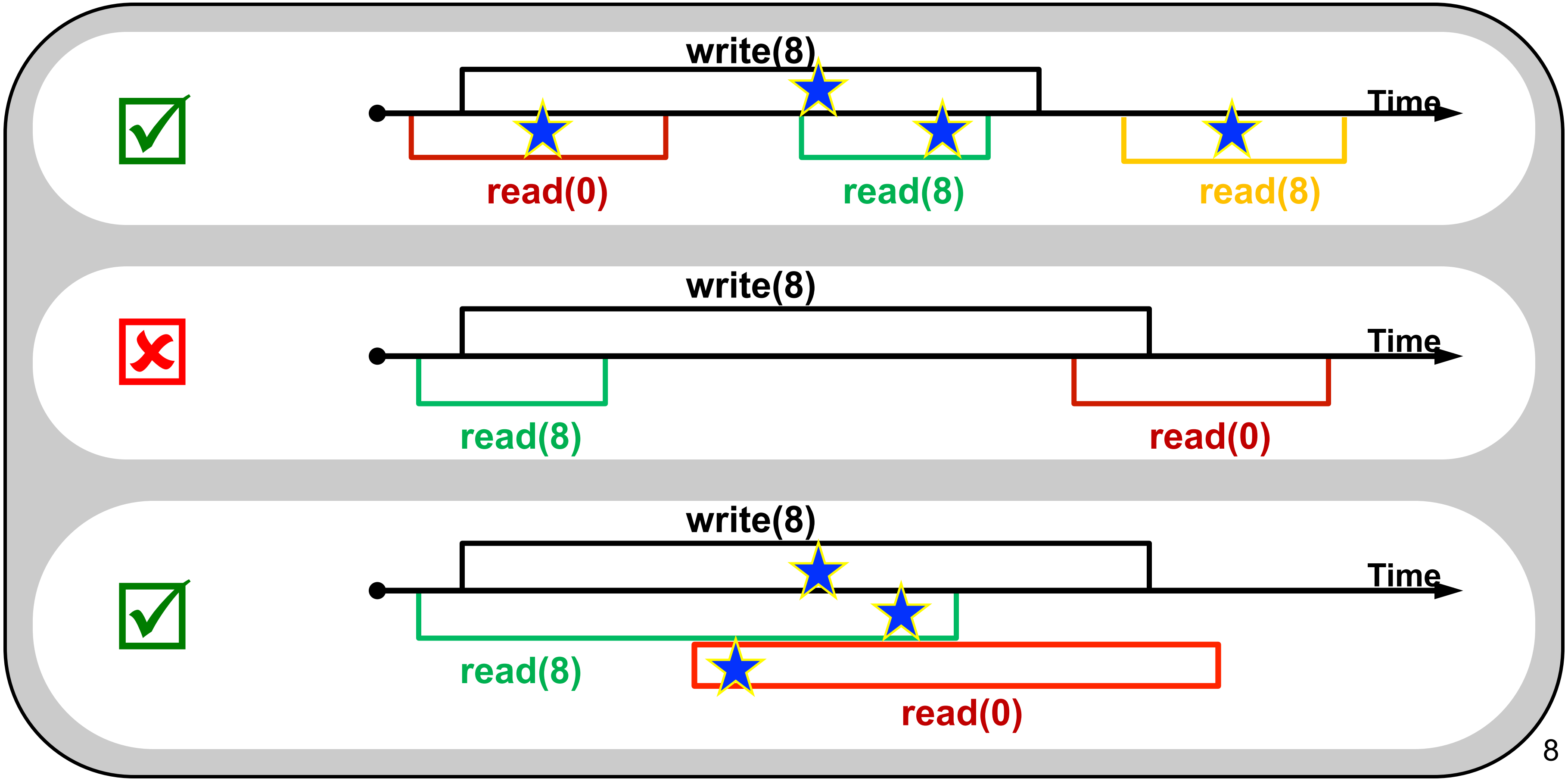
- With replication come challenges:
  - How to preserve consistency while managing replicas?
  - What kind of consistency?
  - How to provide it?
  - How to use it?

# Consistency

- Easiest for users: a single copy view
  - Sequence of operations; a *read* sees the previous *write*
  - **Atomicity** [Lamport] or **linearizability** [Herlihy Wing]
  - Not cheap to implement even without general updates
- Cheapest to implement: a *read* sees a subset of prior *writes*
  - Not the most natural semantic for the users
- Additional complications in *dynamic systems*
  - Ever-changing sets of replicas and participants
  - Crashes never stop, timing variations persist
  - Evolving topology
  - Ultimately mobility

# Atomicity / Linearizability [Lamport / Herlihy Wing ]

- "Shrink" the interval of each operation to a serialization point so that the behavior of the object is consistent with its sequential type



# Consistency Polemics

- Distributed theory focus
  - Fault-tolerance
  - Consistency
- Parallel/Distrib. architectures
  - Performance
  - Speed-up

**Yes,**
**mine is slow…**
**But it is correct!**

**Yes,**
**mine is wrong…**
**But it is fast!**

- User:

**Can’t they get along?**

# Using Majorities/Quorums for Consistency

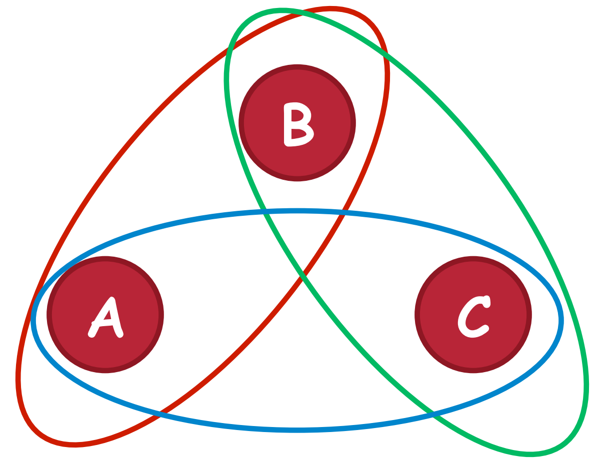
- Consistency of replicated data: using *intersecting sets*
  - Starting with Gifford (79) and Thomas (79)
  - Upfal and Wigderson (85)
    - Majority sets of readers and writers emulate shared memory in a synchronous distributed setting
  - Vitanyi and Awerbuch (86)
    - MW/MR registers using matrices of SW/SR registers where rows and columns are read and written
  - Attiya, Bar-Noy, and Dolev (91/95, 2011 Dijkstra Prize)
    - Atomic SW/MR objects in message passing systems, majorities of processors, ***minority may crash***
    - ***Two-phase protocol (ABD)***

# Related Other Approaches

- Using specialized communication primitives [Imbs, Mostéfaoui, Perrin, Raynal - NETYS 2017]
  - Set constrained delivery broadcast
  - Leading to a snapshot implementation
  - Ultimately atomic read/write objects
- Using *consensus* to agree on each operation [Lamport]
  - Performance overhead for each reads and write op
  - Termination of operations depends on consensus
- Use *group communication* service [Birman 87] with TO bcast [Amir, Dolev, Melliar-Smith, Moser 94], [Keidar, Dolev 96]
  - View change delays reads/writes
  - One change may trigger view formation

# Quorum Systems and Examples

*Quorum system* **Q** over **P**, a set of processor ids:
$\mathbf{Q} = \{Q_1, Q_2, \dots\}$
- $Q_i \subseteq \mathbf{P}$
- $Q_i \cap Q_j \neq \emptyset$ for all $i, j$




**Majorities [Thomas79,Gifford79]**

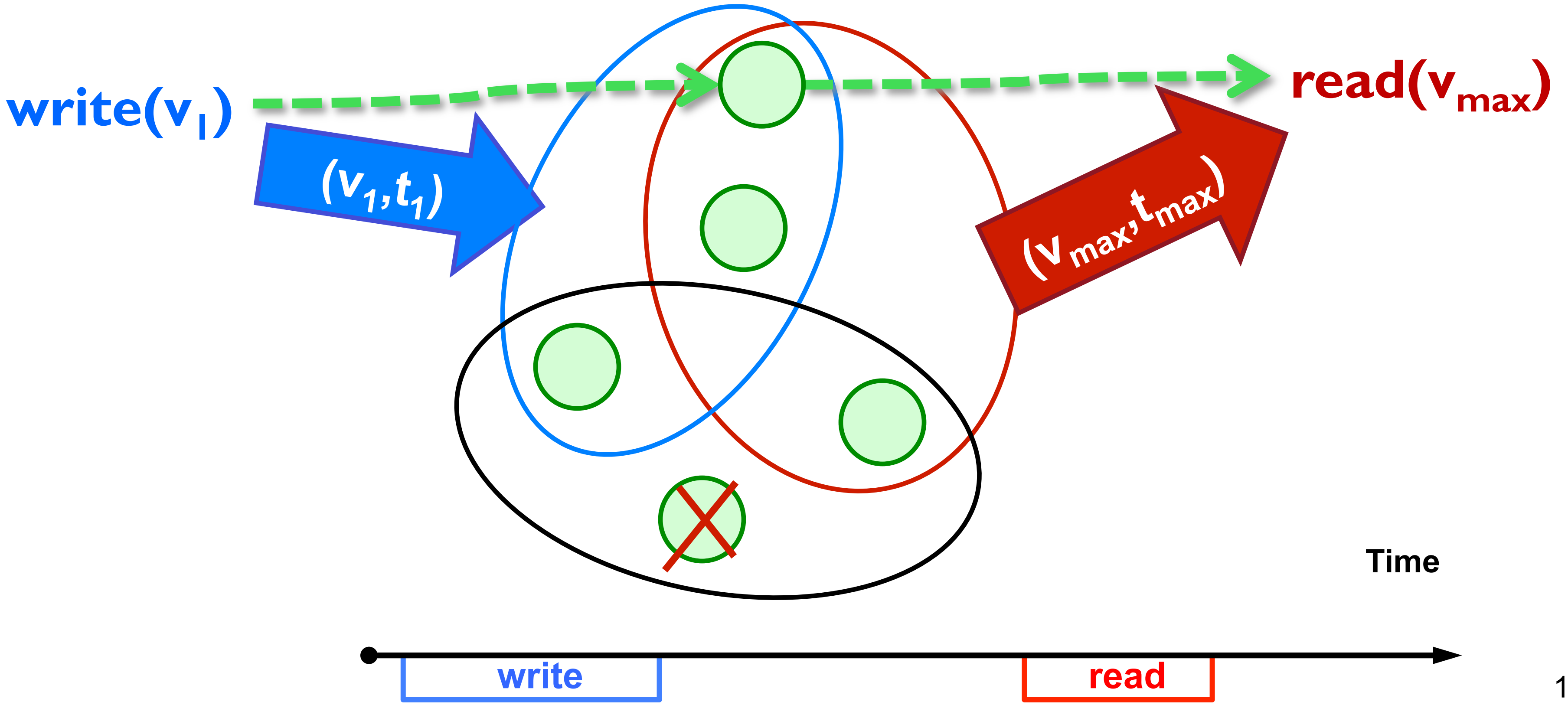
| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

**Matrix Quorums:**
**Processor ids arranged in a matrix.**
**A quorum: Row ∪ Column**

*Lemma:*
The *join* of quorum system $\mathbf{Q}_a$ over $\mathbf{P}_a$ and system $\mathbf{Q}_b$ over $\mathbf{P}_b$, $\mathbf{Q}_a \bowtie \mathbf{Q}_b$, is a quorum system over $\mathbf{P}_a \cup \mathbf{P}_b$.

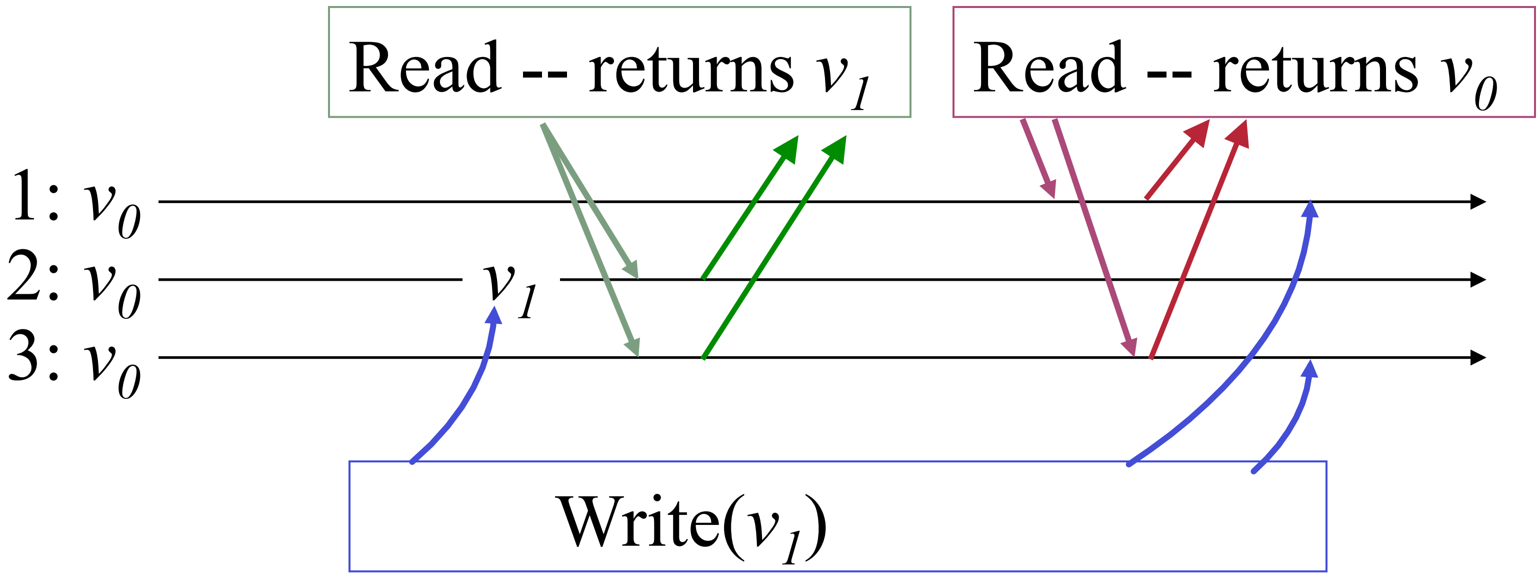
# Main Idea: Timestamps (logical) and Quorums

- An object is represented by a pair (value, timestamp)
- A write records (new-value, new-timestamp) in a quorum
- A read obtains (value, timestamp) pairs from a quorum, then returns the value with the largest timestamp



# يجب على القارئ الكتابة

- If operations are concurrent and a reader simply returns the latest value, then atomicity can be violated:




- Solution: "Readers must write": If readers first help the writer to record the value in a quorum, then it is safe to return the latest value

# The Read Algorithm [ABD]

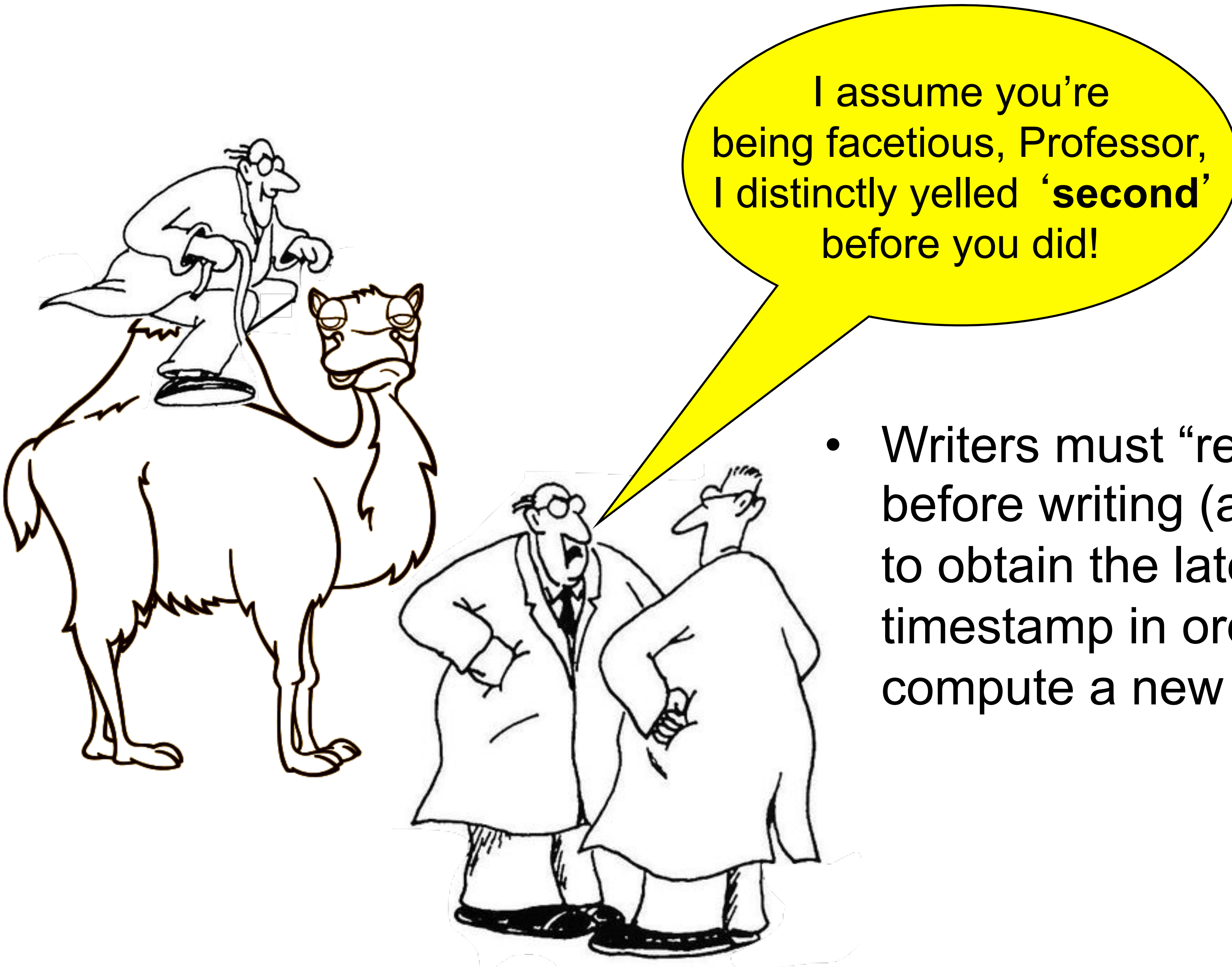
---

$read_i(v : \text{output})$

**Get:** Broadcast $\langle get, i \rangle$ to all replica hosts.
Await responses $\langle get\text{-}ack, val, tag \rangle$ from some majority of replicas.
Let $v$ be the value that corresponds to the maximum tag $maxtag$ received.

**Put:** Broadcast $\langle put, v, maxtag, i \rangle$ to all replica hosts.
Await responses $\langle put\text{-}ack, v, maxtag \rangle$ from some majority of replicas.

# Lastly: “Riders... uhm... Writers Must Read”




- Writers must “read” before writing (and riding) to obtain the latest timestamp in order to compute a new timestamp

# The Complete Algorithm [ABD, as in LS'97]

$read_i(v : \text{output})$

**Get:** Broadcast $\langle get, i\rangle$ to all replica hosts.
Await responses $\langle get\text{-}ack, val, tag\rangle$ from some majority of replicas.
Let $v$ be the value that corresponds to the maximum tag $maxtag$ received.

**Put:** Broadcast $\langle put, v, maxtag, i\rangle$ to all replica hosts.
Await responses $\langle put\text{-}ack, v, maxtag\rangle$ from some majority of replicas.

$write_i(v : \text{input})$

**Get:** Broadcast $\langle get, i\rangle$ to all replica hosts.
Await responses $\langle get\text{-}ack, val, tag\rangle$ from some majority of replicas.
Let $maxtag = \langle seq, pid\rangle$ be the maximim tag received.
Set $newtag = \langle seq + 1, i\rangle$.

**Put:** Broadcast $\langle put, v, newtag, i\rangle$ to all replica hosts.
Await responses $\langle put\text{-}ack, v, newtag\rangle$ from some majority of replicas.

- Read and write uses identical two-phase communication patterns:
- **Get phase**: query and obtain values from a **majority** (quorum),
- **Put phase**: propagate values to a **majority** (quorum).
- The only difference is in what is sent out in the Put phase.

Upon $receive(\langle get, j\rangle)$ at $i$
Send $\langle get\text{-}ack, value_i, tag_i\rangle$ to $j$.

Upon $receive(\langle put, v, t, j\rangle)$ at $i$
If $t > tag_i$ then Set $value_i$ to $v$ and $tag_i$ to $t$.
Send $\langle put\text{-}ack, v, t\rangle$ to $j$.

- Replica hosts respond to Get and Put requests
- Any **minority** may crash

# Latency of Atomic Reads and Writes

- Network latency is key in assessing efficiency
  - Let ***d*** be the max latency (unknown to the algorithm)
  - 1 message exchange incurs delay ***d***
  - 1 round-trip exchange = 2 message exchanges = 2 ***d***
- Single-Writer/Multiple Readers (SWMR)
  - Read latency = 4***d*** : 2 round-trips = 4 exchanges
  - Write latency = 2***d*** : 1 round-trip = 2 exchanges
- Multiple-Writers/Multiple Readers(MWMR)
  - Read latency = 4***d*** : 2 round-trips = 4 exchanges
  - Write latency = 4***d*** : 2 round-trips = 4 exchanges
- **Can we have 2-exchange reads?**

# SWMR: Reads and Writes with 2*d* Latency

- Conditions for enabling ***fast*** operations -- latency 2***d***
  - [Dutta, Guerraoui, Levi, Chakraborty 2004]
- SWMR atomic registers
  - Both reads and writes take 2 exchanges
  - The maximum number of readers ***R*** must be constrained wrt to the number of replica servers ***S***, and the number of server crashes ***F*** : $R < (S/F) - 2$
  - Again, exploiting intersection properties
- Impossibility result for MWMR
  - Fast implementations are impossible when $F \geq 1$

# MWMR: Can some Reads have Latency 2*d*?

- It is possible for reads to terminate early, in 2 exchanges
  - [Dolev, Gilbert, Lynch, S., Welch 2005]
- If after first phase there is a majority of servers reporting the same latest tag (timestamp)
  - Then second phase is unnecessary
- More generally: Maintain a set of ***confirmed*** tags
  - Gossip in the background, or piggyback to messages
  - If a tag is ***confirmed***, then second phase is not needed
- Can one examine the properties of the set of responses and establish conditions under which operations can be ***fast***, i.e., taking 2 exchanges?

# "Semifast" Implementations
## [Georgiou, Nicolaou, S. 06, 09]

- Atomic SWMR memory with unbounded number of readers
  - Group multiple writers into "virtual nodes"
  - Examine the properties of collected server responses
- Results
  - Writes are fast: 2 exchanges (1 round), with latency 2***d***
  - Reads perform 2 or 4 exchanges (1 or 2 rounds), with latency 2***d*** or 4***d***
  - Only a single complete slow read per write operation
    - Any read that precedes or succeeds the slow read and returns the same value is **fast**
  - There exists an execution with only fast operations
  - Holds for $F < S / 3$

# "Weak Semi-Fast" Implementations

- Theorem: [GNS09] It is not possible to devise a MWMR semi-fast implementation even with $W=2$, $R=2$ and $F=1$.
- Define Weak Semi-Fast property
  - Allows multiple slow – latency 4***d*** – reads per write
- Introduce SSO: Server Side Ordering [GNS 2011]
  - Tag is incremented by the servers and not by the writer.
  - Generated tags may be different across servers
  - Clients decide operation ordering based on server responses
- Use algorithms with ***n-wise*** quorums
  - Any ***n*** quorums have non-empty intersection

# “Weak Semi-Fast” Algorithm [GMS11]

- Write: Send v and gather candidate tags from a quorum
  - Exists tag t in > (n/2)--wise intersection
    - YES – assign t to the written value and return - **FAST: 2*d***
    - NO - propagate unique largest tag to a quorum - **SLOW: 4*d***
- Read: Collect list of writes and tags from a quorum
  - Exists max tag t in >(n/2)--wise intersection
    - YES – return the value written by that write - **FAST: 2*d***
    - NO - propagate largest confirmed tag to a quorum - **SLOW: 4*d***
- Simulations show that savings can be substantial
  - Only 15% slow operations in some scenarios

# What about Operations with 3 exchanges?
## [Hadjistasi, Nicolaou, S. -- NETYS'2017]

- Oh-RAM! "One and a half Round Atomic Memory"
- Protocol idea to obtain operations with latency 3***d***
  - 1st exchange: operation invoker contacts servers
  - 2nd exchange: servers gossip
  - 3rd exchange: servers respond to the invoker
- Impossibility of 3 exchange MWMR memory [TNS'17]
  - No atomic implementations exist where all operations use 3 exchanges, even with a single server crash
- Our algorithms

| Model | Read Exch | Write Exch | Read Comm | Write Comm |
|---|---|---|---|---|
| **SWMR** | **2 or 3** | **2** | **$S^2 + 3S$** | **2 S** |
| **MWMR** | **2 or 3** | **4** | **$S^2 + 3S$** | **4 S** |

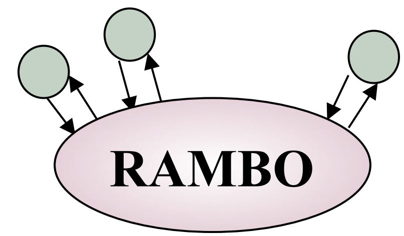
# Dynamic Atomic Memory

- Goal: Atomic Objects in Dynamic Settings
- “Dynamic” encompasses
  - Changing sets of participants: nodes come and go as they please
  - Wide range of failures
  - Asynchrony, timing variations
  - Crashes, message loss, weak delivery guarantees
  - Changes in network topology
  - Processor mobility
- Our solution: RAMBO
  - Reconfigurable Atomic Memory for Basic Objects [Lynch Schwarzmann]

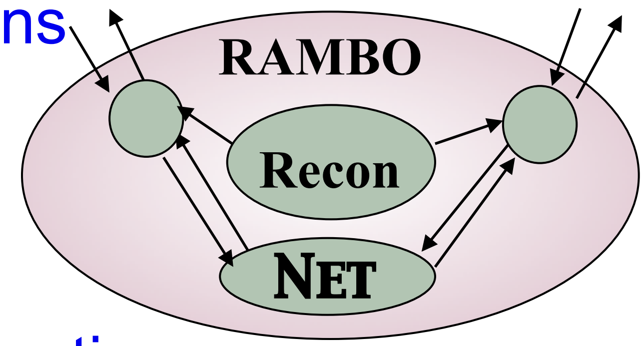
# RAMBO: Approach

- Objects are replicated at several network locations
- To accommodate small, transient changes:
  - Use quorum configurations:  members, quorums
  - Maintains atomicity during "normal operation"
  - Allows concurrency
- To handle larger, more permanent changes:
  - Reconfigure: emit and use new configurations
  - Use consensus to impose total order (Paxos)
  - Maintains atomicity across configuration changes
  - Any configuration can be installed at any time
  - Reconfigure concurrently with reads/writes -- operations ***do not depend*** on view change completion

# Reconfigurable Atomic Memory for Basic Objects



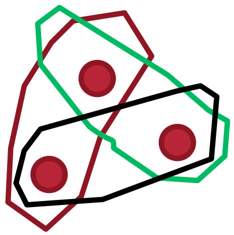

- Global service specification
- Implementation:
  Main algorithm + "recon" service
- Recon service:
  - "Advance reconnaissance"
  - Consistent sequence of configurations
  - Loosely coupled



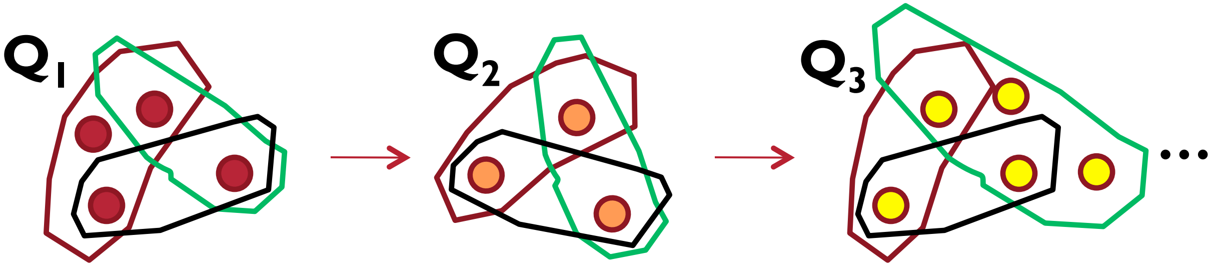

- Main algorithm:
  - Reading, writing
  - Receives, disseminates new configuration information; no explicit installation
  - Reconfigures: upgrade to new and remove old
  - Reads/writes may use several configurations

# Configurations and Reconfiguration

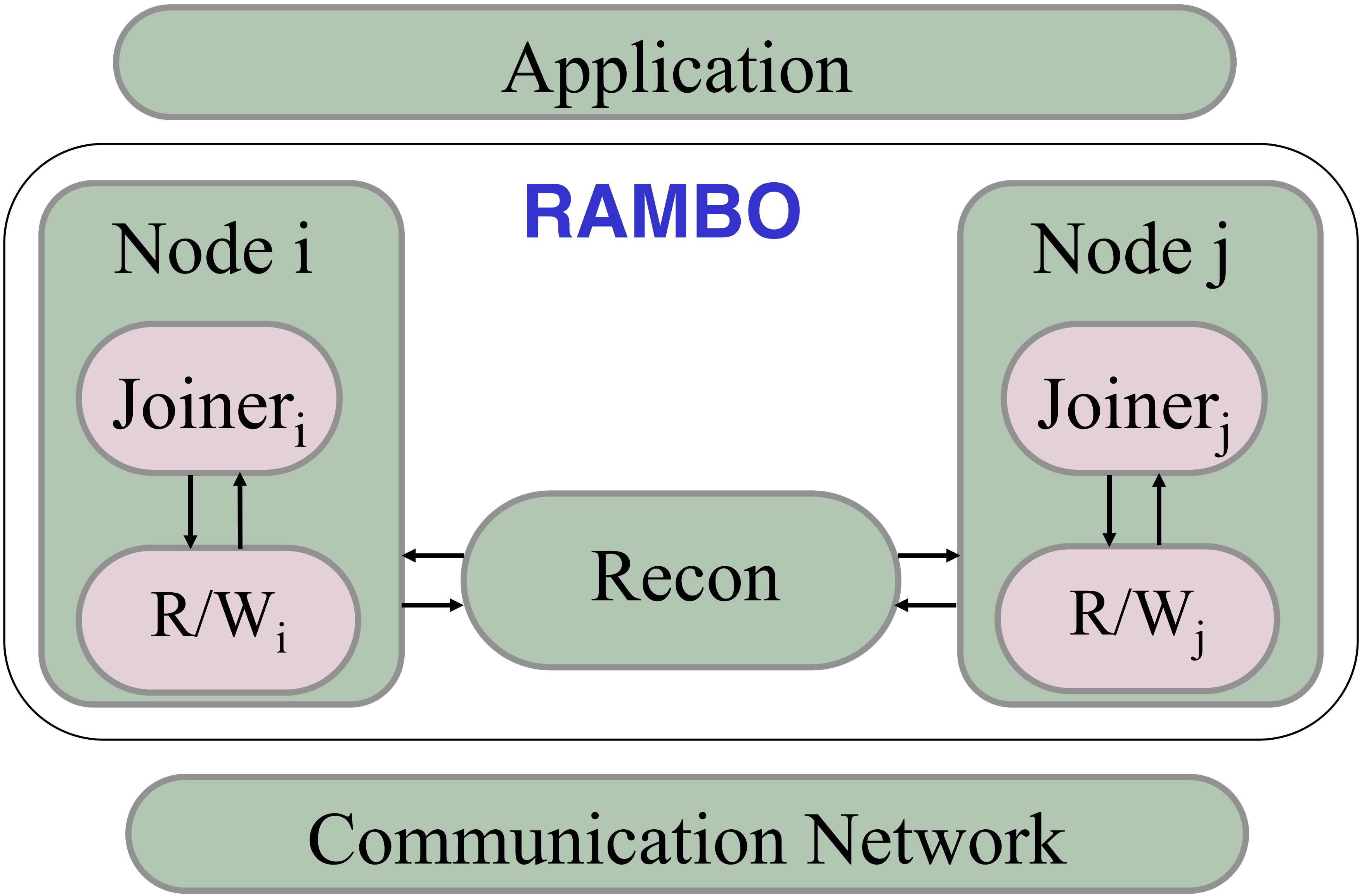
- Configuration: quorum system
  - Collection of subsets of replica host ids where any two subsets intersect
  - (Alternatively: read- and write-quorums, where any read-quorum intersects any write-quorum)



- Reconfiguration process involves two decoupled steps
  - Recon: Emit a new configuration; then later**…**
  - Garbage-collect obsolete configurations locally and "upgrade" to the latest known configuration
  - ***No*** constraints on memberships of quorum systems



# Architectural View

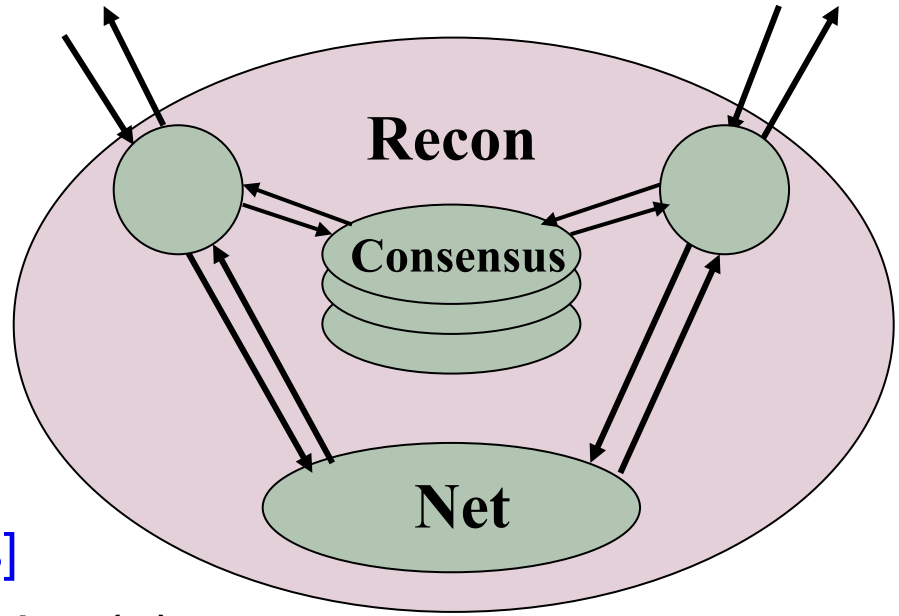


# High-Level Functions

- Joiner
  - Introduces new participants to the system
- Reader-Writer
  - Routine read and write operations
  - Two-phase algorithm using all "known" configurations
  - Using tags to time-stamp (and order) written values
- Recon
  - Chooses new next configuration, e.g., using Paxos
  - Informs the members of the current configuration
- Configuration upgrade ("packaged" with Reader-Writer)
  - Identify and remove obsolete configurations (garbage collection)

# Implementation of Recon

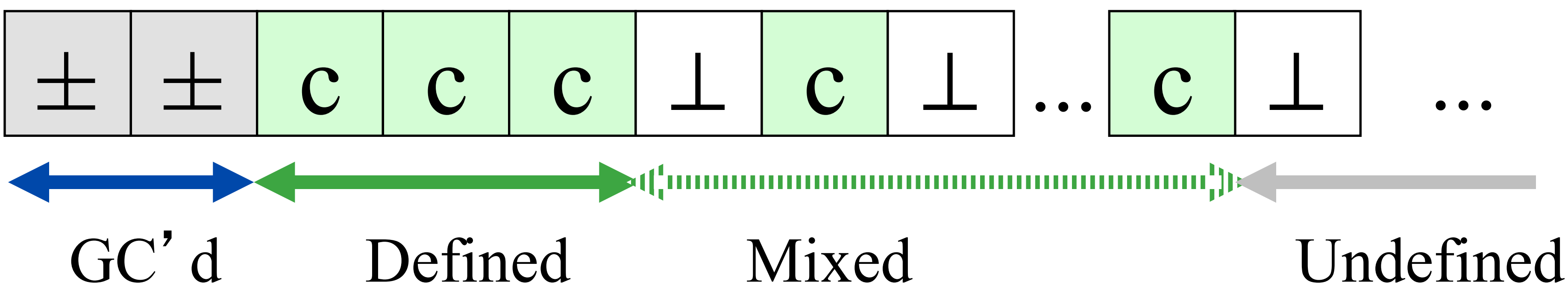
- Uses consensus to determine new configurations 1,2,3,…
  - Note: when the universe of configurations is finite and known, then consensus is ***not*** needed even with unbounded reconfiguration [GeoQuorums]



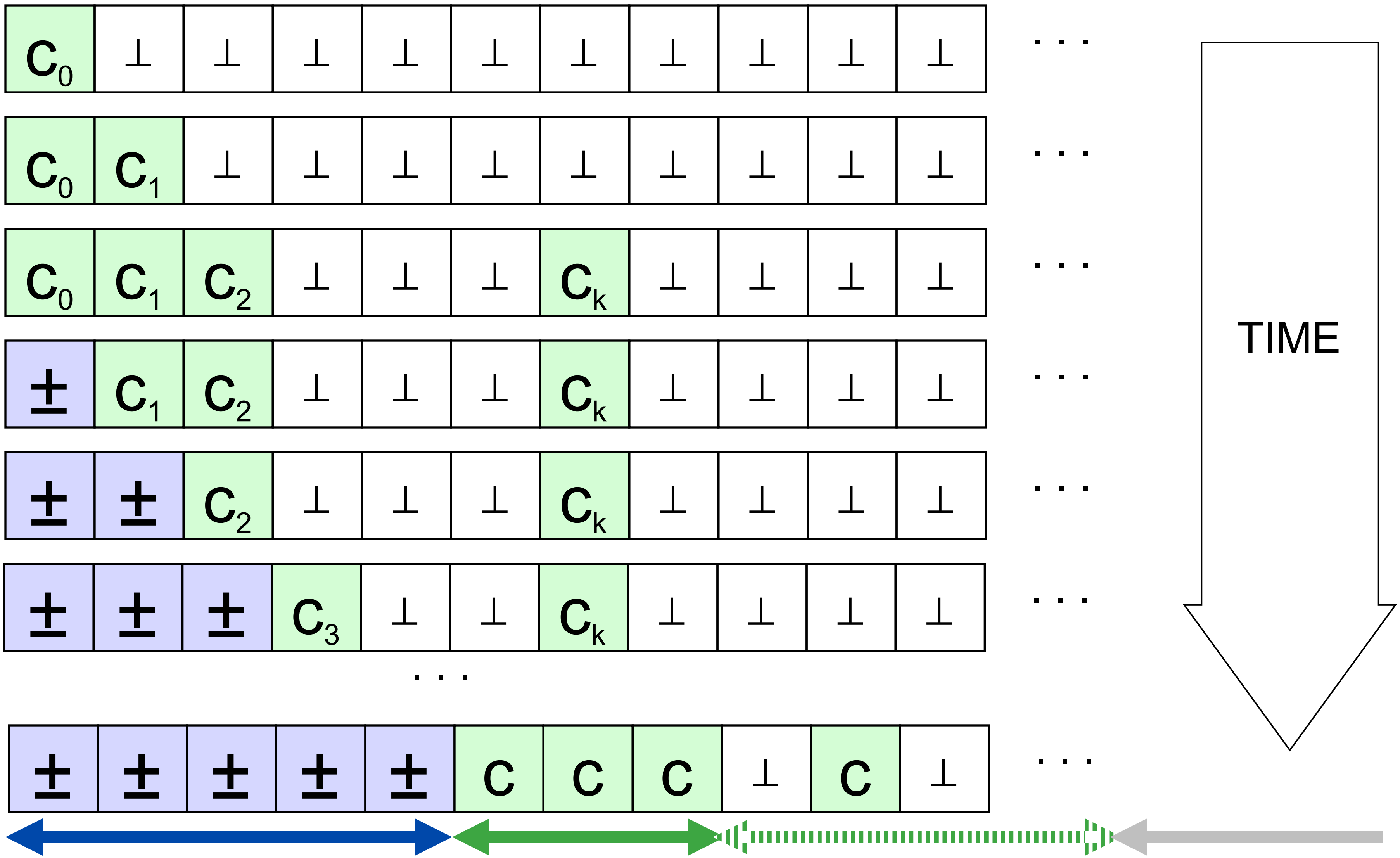

- Members of existing configuration(s) may propose a new configuration
- Proposals reconciled using consensus
- Consensus is a fairly heavy mechanism, but it is
  - Used only for reconfigurations, which are infrequent
  - Does ***not*** block or abort Read and Write operations

# Configurations and Config Maps

- Configuration c
  - *members*(c) -- set of members of configuration c
  - *read-quorums*(c), *write-quorums*(c) -- sets of quorums
- Configuration map cm
  - mapping from naturals to configurations
  - cm(*k*) is configuration *k*
  - Can be defined (c), undefined (⊥), garbage-collected (±)



# Configuration Map Changes (Local View)



# Configuration Upgrade [Gilbert, Lynch, S 10]

- Reconfigure to last configuration in a contiguous segment

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | . . . | ± | $c_j$ | . . . | $c_k$ | . . . | | ⊥ | . . . |

- Phase 1:
  - Informs write-quorum of $c_j$ … $c_{k-1}$ about $c_k$
  - Collects (value,tag) from read-quorums of $c_j$ … $c_{k-1}$
- Phase 2:
  - Propagates latest (value, tag) to a write-quorum of $c_k$
  - Garbage-collect: Set $cmap(j \ldots k-1)$ to ±
- Constant-time upgrade regardless of the number of obsolete configurations (conditioned on failures)
- Maintains good read/write latency during system instability or frequent reconfigurations

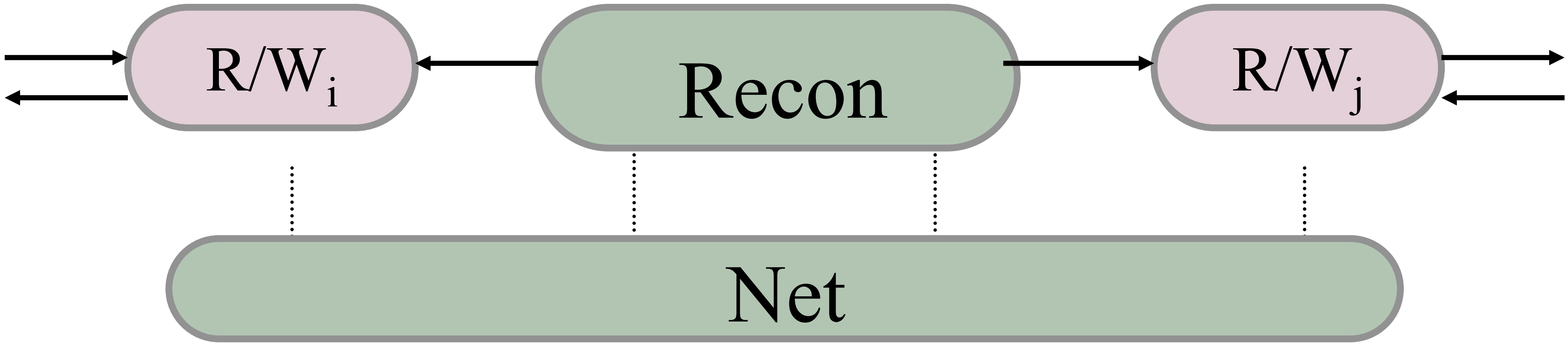
# On to Reads and Writes: Values and Tags

- Each value ***v*** has an associated tag ***t*** (logical timestamp)
  - *Tag* is made up of the *sequence-processor* pair
- Reads:
  - a set of *value-tag* pairs is obtained
  - the result is the *value* with the maximum *tag*
- Writes:
  - a set of *value-tag* pairs is obtained
  - *new-value* is propagated with a *new-tag* that is a lexicographic increment of *tag* :

$$new\text{-}tag := \langle tag.seq + 1, pid \rangle$$

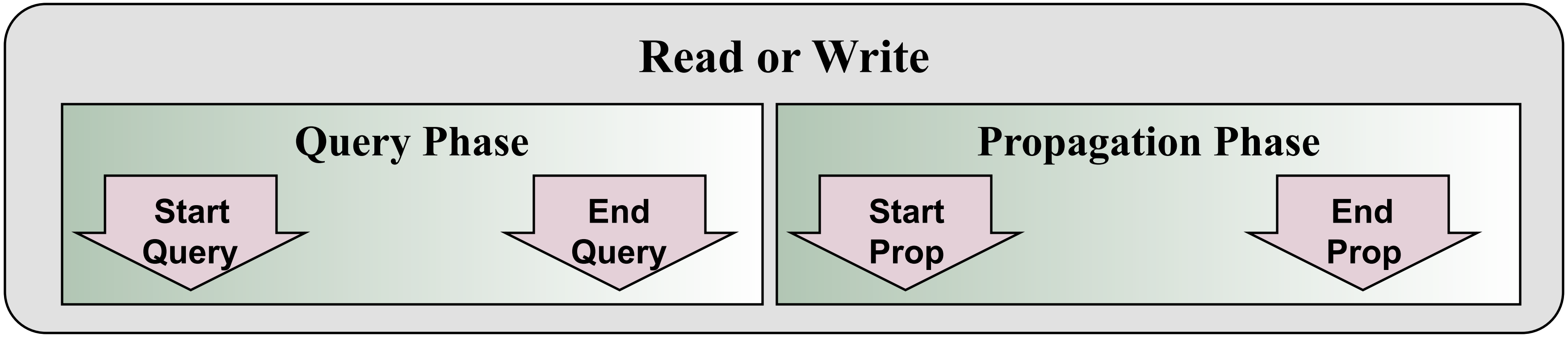
# Dynamic Reader-Writer and Recon

- The work is split between Reader-Writer and Recon
- Recon emits consistent configurations
- Reader-Writer processes run two-phase quorum-based algorithm, with all "active" configurations
- Background "gossip" builds fixed-points
- If Recon emits new configuration, Reader-Writer continues reads/writes in progress, until fixed-point is reached

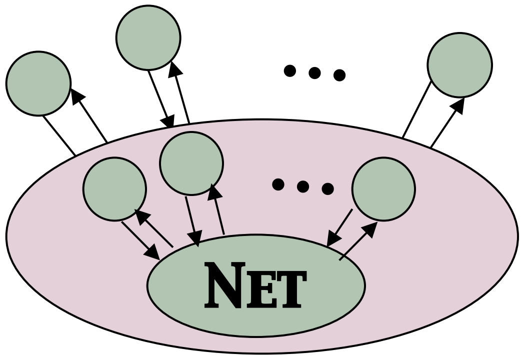


# Processing Reads and Writes

- Reads and Writes perform Query and Propagation phases using known configurations, stored in *op.cmap.*
  - Query: Gets latest *value*, *tag*, and *cmap* information from read-quorums
  - Propagation: Gives latest *(value,tag)* to write-quorums
  - Both phases: Extend *op.cmap* with newly-discovered configurations that now must also be involved.
- Each phase ends with a **fixed point**, involving all the configurations currently in *op.cmap*



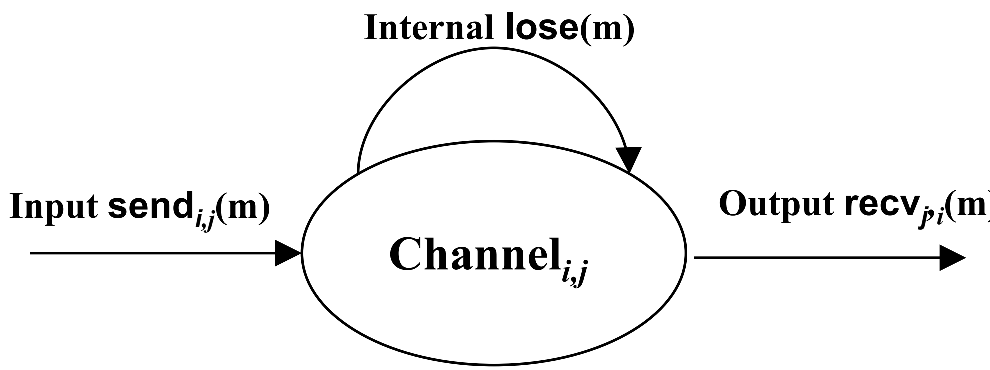
# Methodology




- Algorithms are presented formally, using interacting state machine models: Input/Output automata
  - service specifications
  - algorithm descriptions
  - models for applications
- Safety: rigorous proof of correctness (atomicity) for arbitrary patterns of asynchrony and change
- Conditional performance analysis
  - E.g., when message latency $\leq$ ***d***, quorum configurations are "viable", then read and write operations take time between 4***d*** and 8***d***, under reasonable "steady-state" assumptions.

# Example Spec: Asynchronous Lossy Channel

- Input Output Automata [Lynch & Tuttle]
- Supports: composition, abstraction, rigorous reasoning
- 100's algorithms




**Domains:**

$I$, a set of processes, $M$, a set of messages

**States:**

$S \subseteq M$, the set of messages in the channel

**Signature:**

| | |
|---|---|
| Input: | $\mathsf{send}(m)_{i,j},\ m \in M$, const $i, j \in I$ |
| Output: | $\mathsf{recv}(m)_{j,i},\ m \in M$, const $i, j \in I$ |
| Internal: | $\mathsf{lose}(m),\ m \in M$ |

**Transitions:**

Input $\mathsf{send}(m)_{i,j}$
Effect:
$S \leftarrow S \cup \{m\}$

Output $\mathsf{recv}(m)_{j,i}$
Precondition:
$m \in S$
Effect:
$S \leftarrow S - \{m\}$

Internal $\mathsf{lose}(m)$
Precondition:
$m \in S$
Effect:
$S \leftarrow S - \{m\}$

# Details: Reader-Writer: Send and Recv Code

Send

Output $\mathsf{send}(\langle W, v, t, cm, ni, nj \rangle)_{i,j}$
Precondition:
$status = active$
$j \in world$
$\langle W, v, t, cm, ni, nj \rangle =$
$\langle world, value, tag, cmap, phase\text{-}num(i), phase\text{-}num(j) \rangle$
Effect:
none

Receive

Input $\mathsf{recv}(\langle W, v, t, cm, nj, ni \rangle)_{j,i}$
Effect:
if $status \neq idle$ then
$status \leftarrow active$
$world \leftarrow world \cup W$
if $t > tag$ then $(value, tag) \leftarrow (v, t)$
$cmap \leftarrow update(cmap, cm)$
$phase\text{-}num(j) \leftarrow \max(phase\text{-}num(j), nj)$
if $op.phase \in \{query, prop\}$ and $ni \geq op.phase\text{-}num$ then
$op.cmap \leftarrow extend(op.cmap, truncate(cm))$
if $op.cmap \in Truncated$ then $op.acc \leftarrow op.acc \cup \{j\}$
else $op.acc \leftarrow \emptyset$
$op.cmap \leftarrow truncate(cmap)$
if $gc.phase \in \{query, prop\}$ and $ni \geq gc.phase\text{-}num$ then
$gc.acc \leftarrow gc.acc \cup \{j\}$

Specification of gossip using Input/Output Automata of [Lynch Tuttle]

# Details: Reader-Writer Fixed Points

Internal query-fix$_i$
Precondition:
$status = active$
$op.type \in \{read, write\}$
$op.phase = query$
$\forall k \in \mathbb{N}, c \in C : (op.cmap(k) = c)$
$\Rightarrow (\exists R \in read\text{-}quorums(c) : R \subseteq op.acc)$
Effect:
if $op.type = read$ then
$op.value \leftarrow value$
else
$value \leftarrow op.value$
$tag \leftarrow \langle tag.seq + 1, i \rangle$
$pnum\text{-}local \leftarrow pnum\text{-}local + 1$
$op.pnum \leftarrow pnum\text{-}local$
$op.phase \leftarrow prop$
$op.cmap \leftarrow cmap$
$op.acc \leftarrow \emptyset$

Phase 1 fixed point

Phase 2 fixed point

Internal prop-fix$_i$
Precondition:
$status = active$
$op.type \in \{read, write\}$
$op.phase = prop$
$\forall k \in \mathbb{N}, c \in C : (op.cmap(k) = c)$
$\Rightarrow (\exists W \in write\text{-}quorums(c) : W \subseteq op.acc)$
Effect:
$op.phase = done$

Specification of fixed points using Input/Output Automata

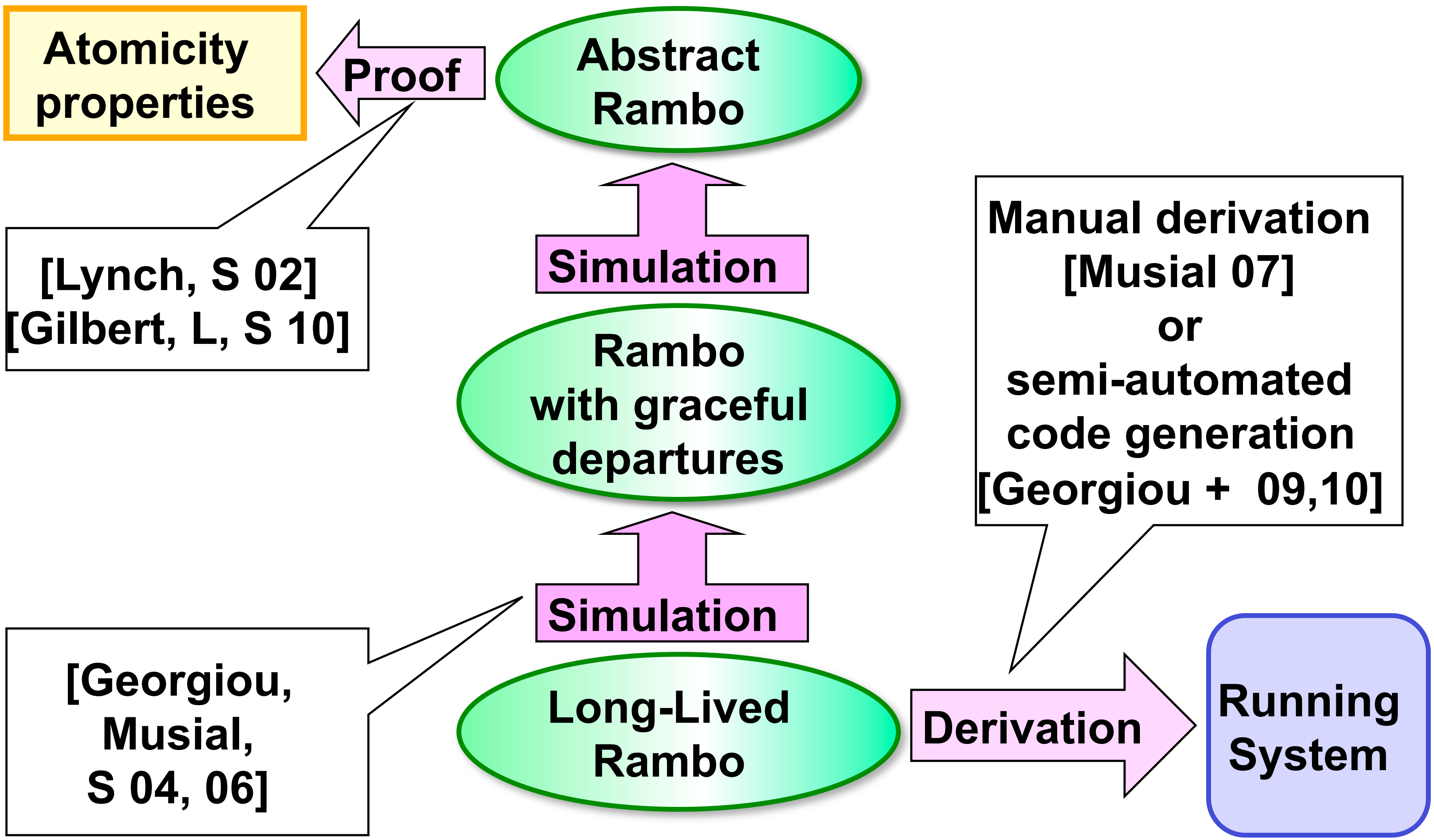
# Some Latency Analysis Results

- Restrict attention to a subset of timed executions
  - Reminder: Read and write operations are ***not*** affected by Recon delays or Recon non-termination
- Configuration upgrade (garbage collection) takes 4***d***
- If reconfigurations are "rare" -- operations take 4***d***
- If configurations are in "steady state" -- operations take 8***d***
- Logarithmic in number of configurations time "catch-up" after a burst of "bad timing behavior"
  - A recovering node joins quickly after a long absence

# Implementation

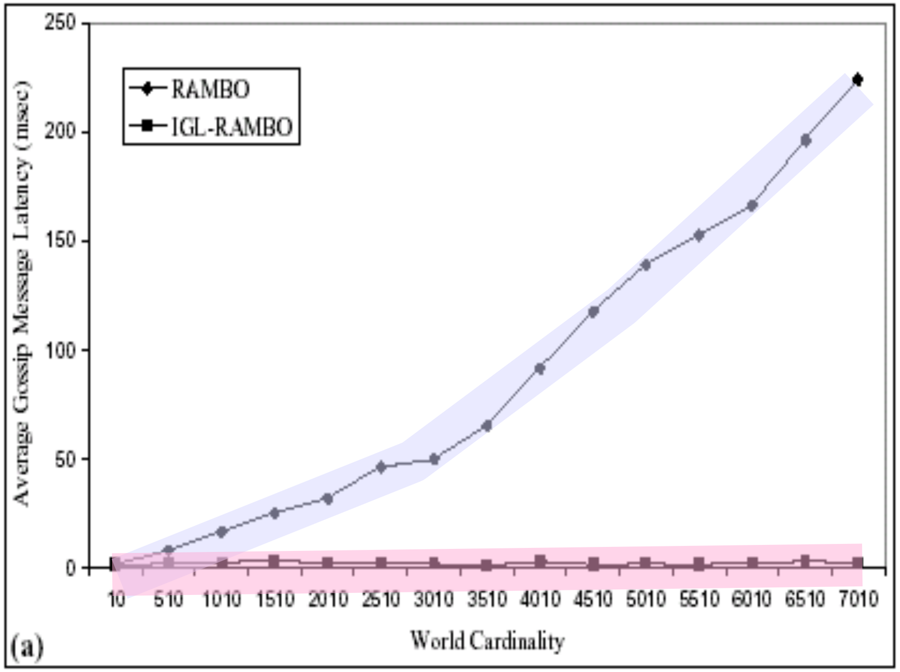
- Experimental system implementations [Musial 07]
  - Platform for refinement, optimization, tuning
  - Observe of algorithms in a local area setting
  - Cluster with 16+/- Linux machines & fast switch
- Developed by manually translating the Input/Output Automata specification to Java code
  - Precise rules are followed to mitigate error introduction during translation
  - Rigorous proofs [Georgiou, Musial, S., Sonderegger 07, 11]
- Next steps:
  - Specification in *Tempo* [Lynch Michel S 08] (Timed IOA)
  - Code generation ([Georgiou Lynch Mavrommatis Tauber 09])

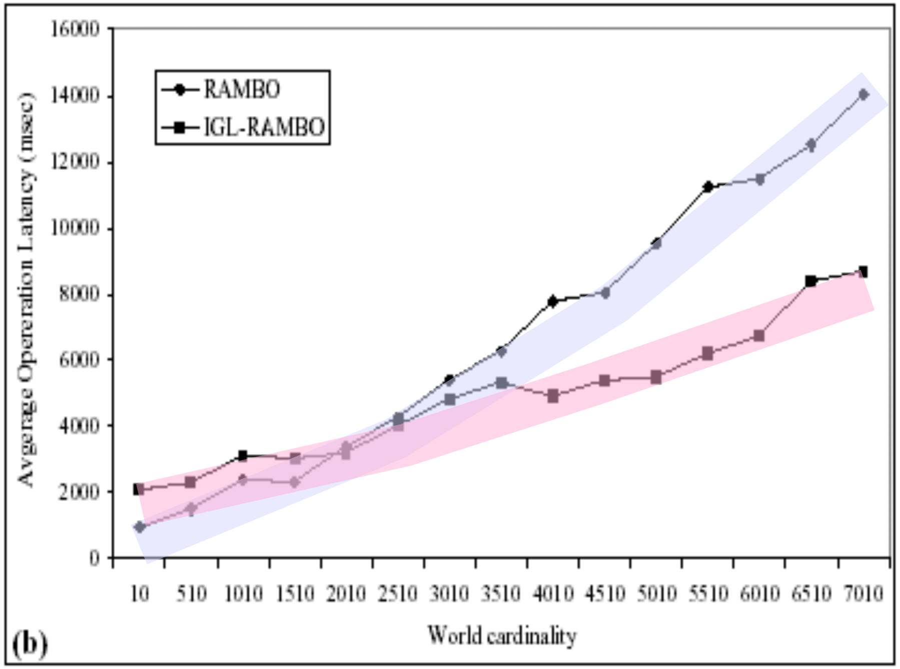
# Optimization and Development Methodology



# Optimization: Improving performance

- Long-Lived RAMBO: Graceful Leave + Incremental Gossip
  - Rigorous proof of correctness by simulation
  - Performance study

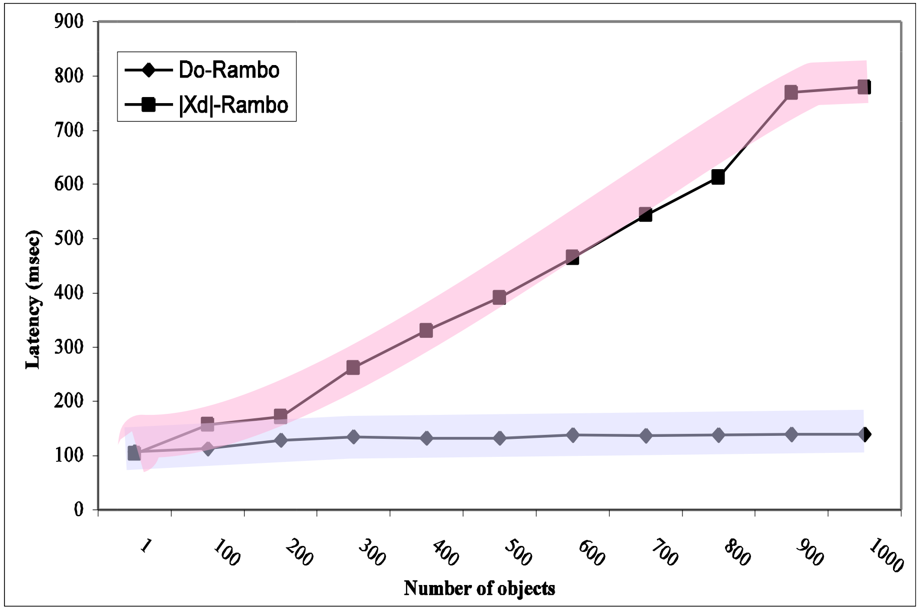



- [Georgiou, Musial, S. 06]

# Complete Shared Memory

- Atomicity is compositional
  - Implement a single memory location
  - Get a complete shared memory by running several implementations: correct, but very slow!
- Domain-oriented reconfigurable atomic memory
  - Optimizing performance for groups of related objects




**[Georgiou, Musial, S. 2009]**

**- Composition**

**- Domain**

# Federated Array of Bricks (FAB)

- Storage system developed and evaluated at HP Labs
  - [Saito Frølund Veitch Merchant Spence 05]
- Distributes workload and handles failures and recoveries without disturbing client requests
- Read or write protocol involves majority quorums of storage "bricks" following the Rambo algorithm
- Evaluations of the implementation showed
  - FAB performance is similar to centralized solutions,
  - While offering at the same time continuous service and high availability

# Additional Solutions

- Atila: *Atomicity Through Indirect Learning Algorithm*
  - Indirect learning enables progress without routing or complete connectivity [Konwar, Musial, Nicolaou, S. 07]
- RDS [Chockler, Gilbert, Gramoli, Musial, S. 09]
  - Reconfigurable Distributed Storage: Rambo ⊕ Paxos
  - Integrate configuration upgrade with installation
  - Obsolete configuration are removed quicker
- DynaStore: Reconfiguration without consensus [Aguilera, Keidar, Malkhi, Shraer 11]
  - Initial quorum system, incremental adds/removes
  - Changes yield DAGs of possibilities
  - Reads/writes use ABD-like phases, traverse DAGs
  - Termination: assumes finite reconfigurations

# DynaDisk Implementation

- Data-center read/write storage system
  - Allows add/remove of storage devices on-the-fly
  - Based on DynaStore, but with and without consensus
  - [Shraer Martin Malkhi Keidar 10]

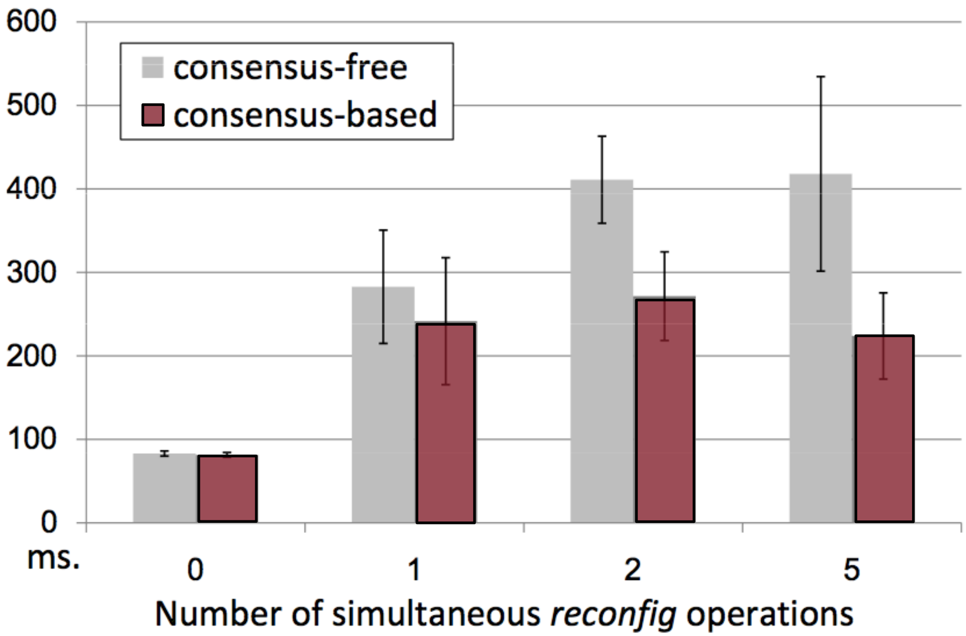



**Figure 1: Average *write* latency.**

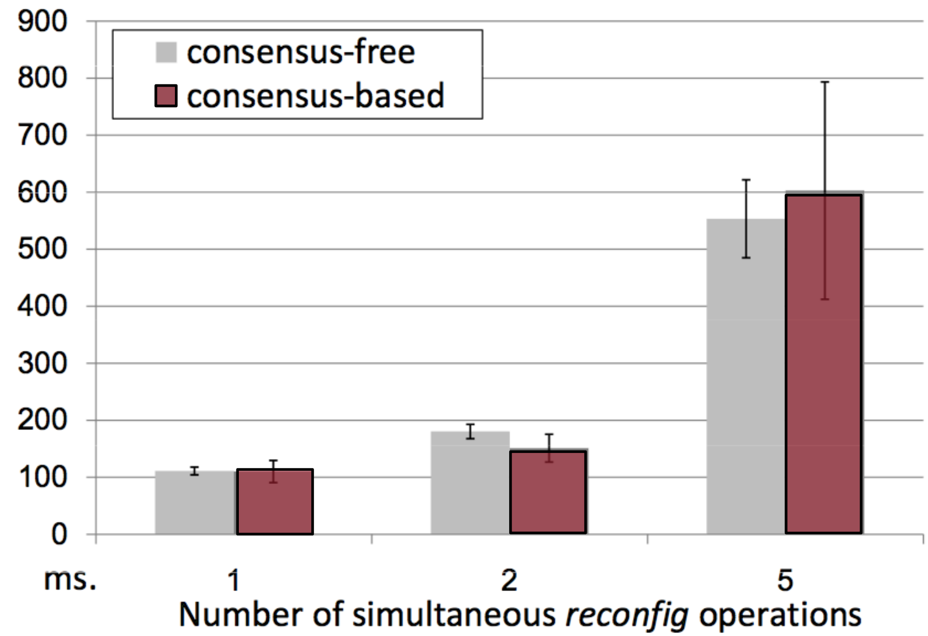


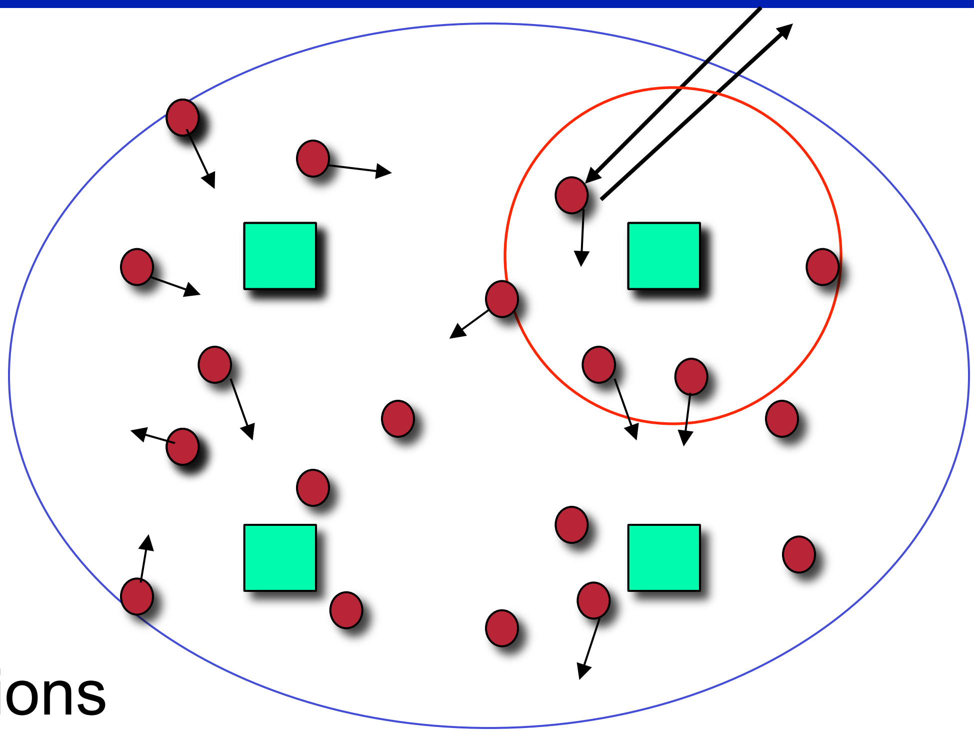

**Figure 2: Average *reconfig* latency.**

# GeoQuorums

- Dynamic atomic read/write memory for mobile settings
  - [Dolev, Gilbert, Lynch, S., Welch 04, 05]
  - Use Rambo architecture over Virtual Node layer
- Nodes: fixed geographical locations called Focal Points
  - Centers of populated, compact geographical areas:
    - Traffic intersections, buildings, bridges, points-of-interest
  - Continuously populated, thus able to maintain state
- Implementations:
  - Virtual Node layer over the physical mobile network
  - Atomic read/write memory over the Virtual Node layer

# GeoQuorums

- Mobile nodes
- Focal points – implemented as Virtual Nodes
- Quorums are defined over focal points
- Use GPS as timestamps
- Fast(er) read/write operations
  - Single phase writes - two exchanges
  - One or two phase reads - two or four exchanges
- Simplified, consensus-free, reconfiguration
  - Two-phase algorithm using fixed configurations
  - Can be motivated by performance: e.g., if writes are frequent, install smaller write quorums



# Closing Remarks: Read-Modify-Write

- RMW is strictly stronger than atomic read/write object
- Some storage systems implement atomic RMW operations
  - Expensive, and requires at its core atomic updates
- Examples
  - Reduce parts of the system to a single-writer model
    - e.g., Microsoft’s Azure
  - Depend on clock synchronization hardware
    - Google’s Spanner
  - Rely on complex mechanisms for resolving event ordering such as vector clocks
    - Amazon’s Dynamo

# Thank You!

## Questions and Discussion