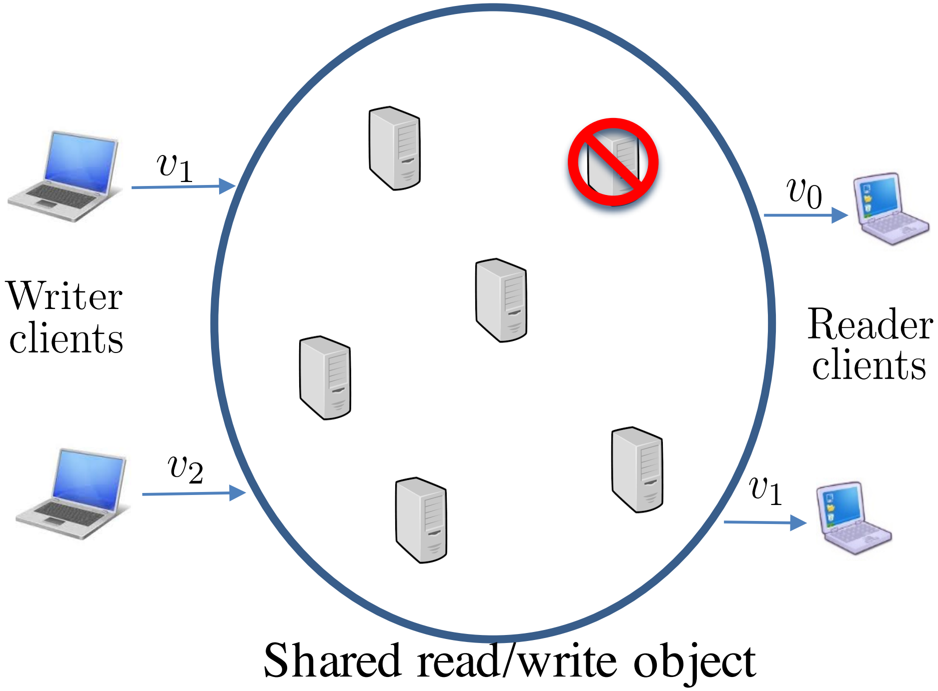
CoVer-ability: Consistent Versioning in Asynchronous, Fail-Prone, Message-Passing Environments

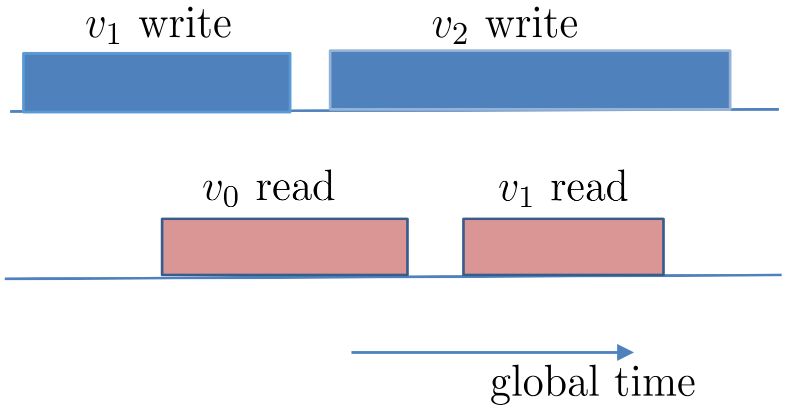
Nicolas Nicolaou , Antonio Fernández Anta, and Chryssis Georgiou

Developing the Science of Networks

Nicolas Nicolaou

Distributed Read/Write Objects



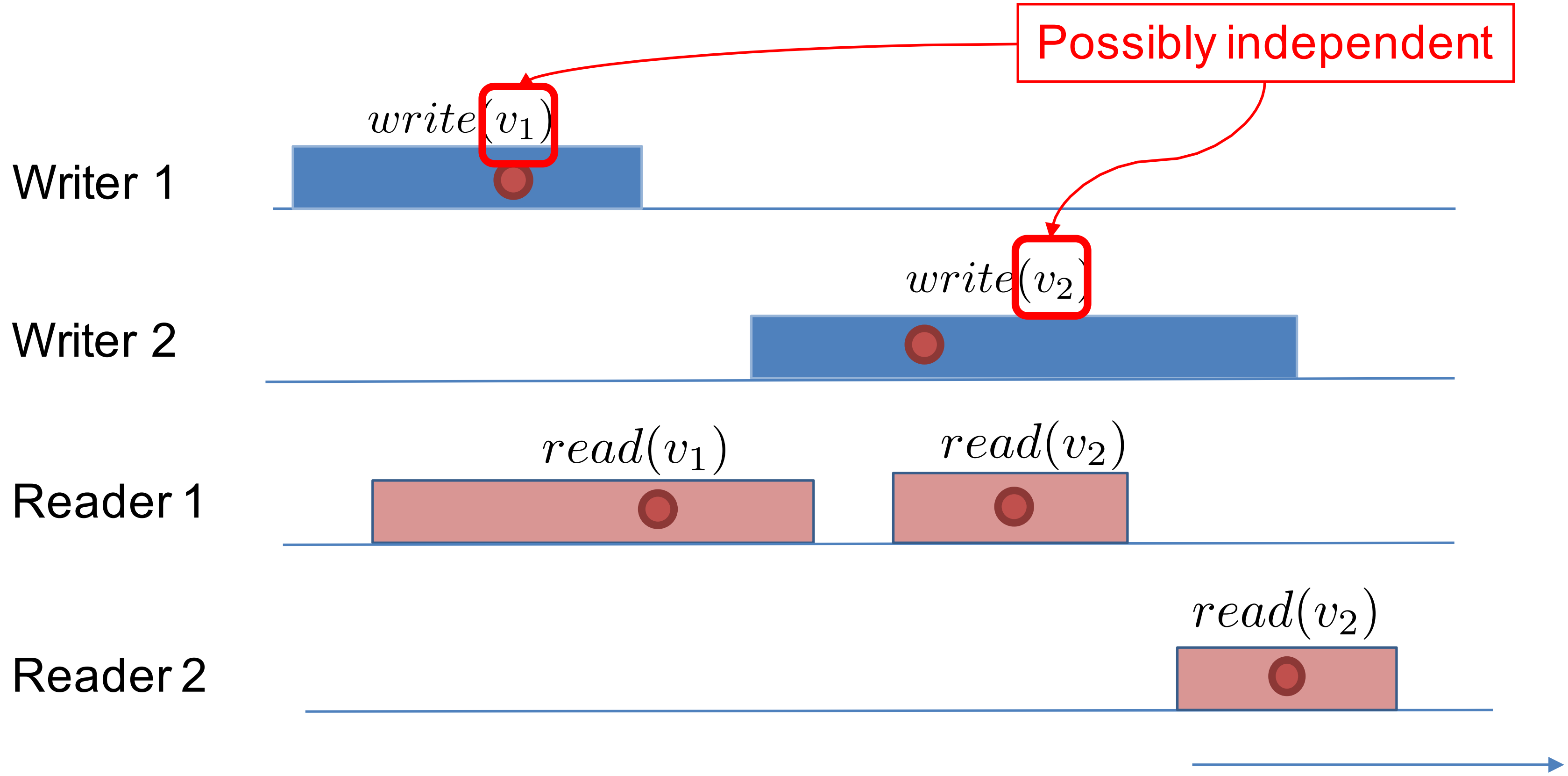


Implementing a fault-tolerant shared object in an asynchronous, message-passing environment:

- Availability + Survivability => use redundancy
- Asynchrony + Redundancy => concurrent operations
- Behavior of concurrent operations => consistency semantics
  - Safety, Regularity, Atomicity [Lamport86]

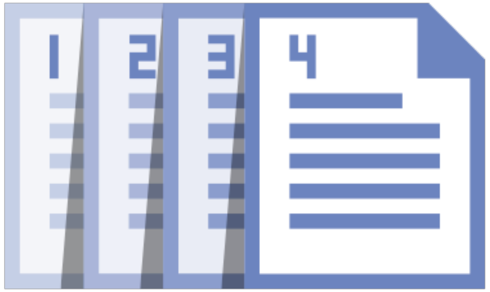
Consistency Model: Atomicity/Linearizability

- 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



Consistency on Object Evolution

- Particular objects may need consistent state evolution
  - File objects
  - Distributed databases
  - Bulleting boards



- Expected Behavior: If a write w1 completes and writes version *i*, then any write w2 that succeeds w1 must write a version *j* s.t.
  - version *j* obtained by modifying version *i*
  - version *j* obtained by modifying version *k* and there is a sequence of modifications that lead from *i* to *k*.

Not preserved with existing consistency semantics

How can we achieve such guarantees?

- Standard practices use:
  - Compare and Swap (CAS)
  - Linked-Load / Store-Conditional (LL/SC)

- Such primitives can solve consensus
  - Impossible in the asynchronous, message-passing, fail prone environment [FLP85]

Question:
Can we provide versioning guarantees in an asynchronous, message-passing, fail-prone environment using weaker primitives, like read/write registers?

Nicolas Nicolaou

System Model: Definitions

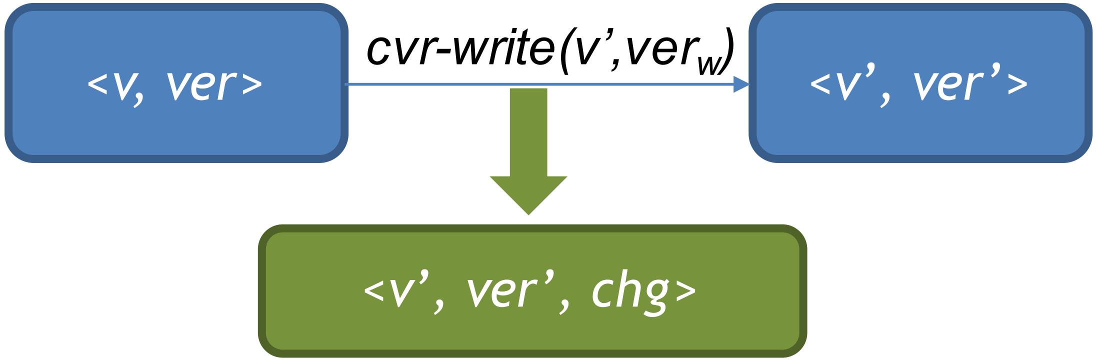
Components	• *n* processes • Asynchronous Computation
Communication	• Asynchronous • Message-Passing • Reliable Channels (messages are not lost or altered)
Failures	• Crashes

Object Type: Versioned Read/Write Register

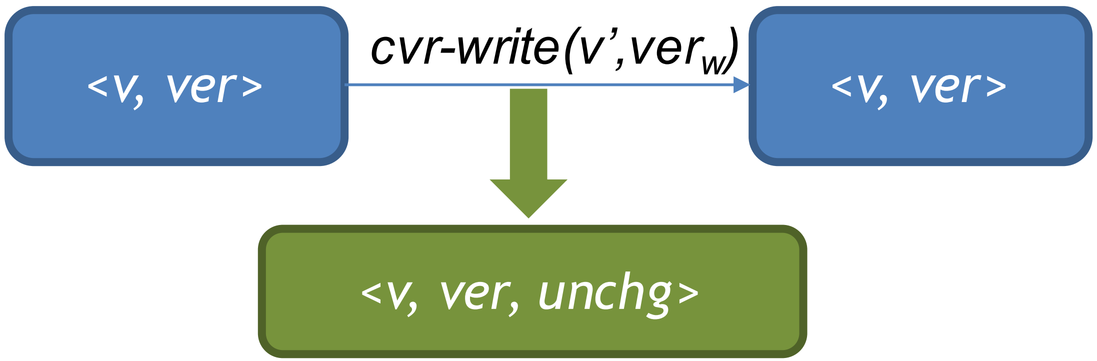
- *Versions*: a totally ordered set of versions
- Domain of Values: *Values x Versions*
  - Each value is assigned with a version
- R/W Register Operations
  - write(*v*): updates the object value to *v*
  - read(): retrieves the object value
- Versioned R/W Register Operations
  - cvr-write(v,ver): attempt to change the value of the object with version ver and returns whether successful or not
  - cvr-read(): retrieves the object value and version

Versioned Register Operations

- Let *<val, ver>* be the state of the register
- Write operations on the versioned register are:
  - Successful write given that $ver_w = ver$:



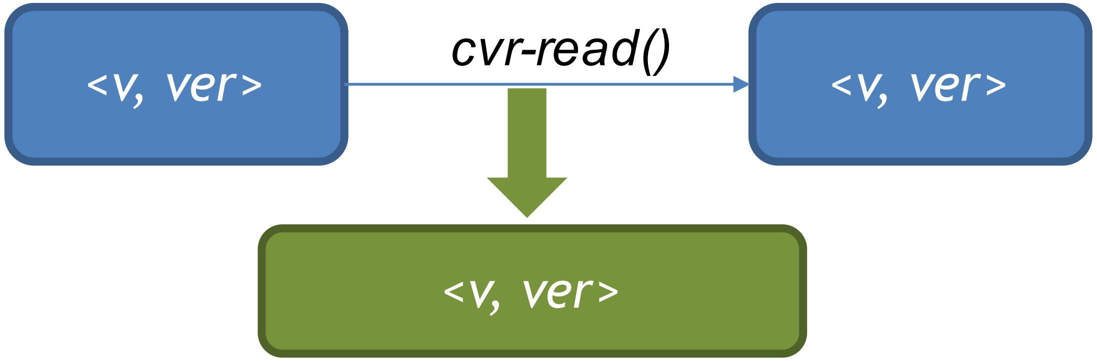
  - Unsuccessful write given that $ver_w \neq ver$:



Nicolas Nicolaou

Versioned Register Operations

- Read operations on the versioned register:



- Notation we use:
  - Successful write: *cvr-w(ver)[ver']*
  - Unsuccessful write: *cvr-w(ver)[ver',unchg]*

Definition: Execution Validity

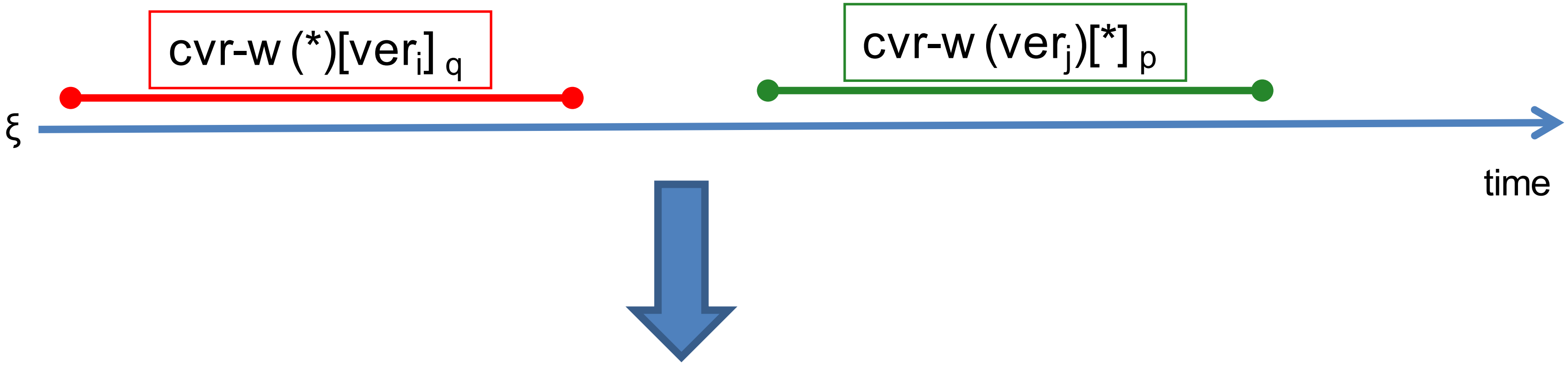
- Version increment
  - For any successful write *cvr-w(ver)[ver']*, *ver' > ver*
- Version uniqueness
  - For any two successful writes *crv-w(*)[ver']* and *crv-w(*)[ver'']*, *then ver'* ≠ ver''
- Version connection
  - For each successful write *cvr-w*$(ver_{k-1})[ver_k]$, there is a sequence of versions ver_0 , ver_1,…,ver_k s.t. there is a successful write *cvr-w*$(ver_i)[ver_{i+1}]$, for $0<i<k-1$

Definition: Coverability

- An execution is coverable with respect to some order $<_\xi$ if it is valid and it satisfies:
  - Version Consolidation
  - Version Continuity
  - Version Evolution

Consolidation in Execution ξ

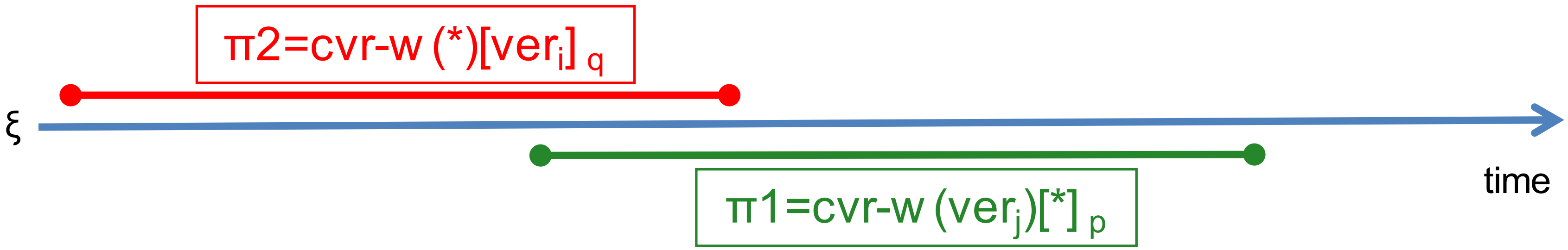
- If two successful writes $\pi 1$ = *crv-w(*)[ver$_i$]* and $\pi 2$=*crv-w(ver$_j$)[*]*, *s.t.* $\pi 1 \rightarrow \pi 2$ in ξ, then $ver_i \leq ver_j$ and $\pi 1 <_\xi \pi 2$



$ver_i \leq ver_j$

Continuity in Execution ξ

- If $\pi 1 = crv\text{-}w(ver_j)[*]$ is a successful write, then there exists successful write $\pi 2 = crv\text{-}w(*)[ver_i]$, s.t. $\pi 2 <_{\xi} \pi 1$ and $ver_i = ver_j$, or $ver_j = ver_0$



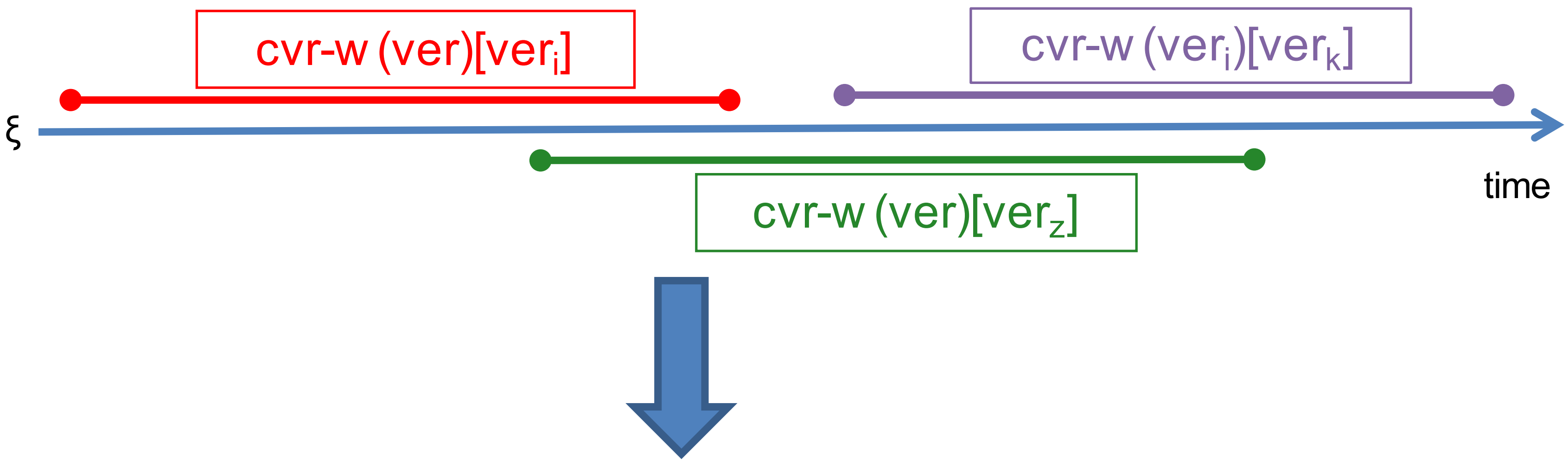
$\pi 2 <_{\xi} \pi 1$ <u>AND</u> $ver_i = ver_j$

OR

$ver_j = ver_0$

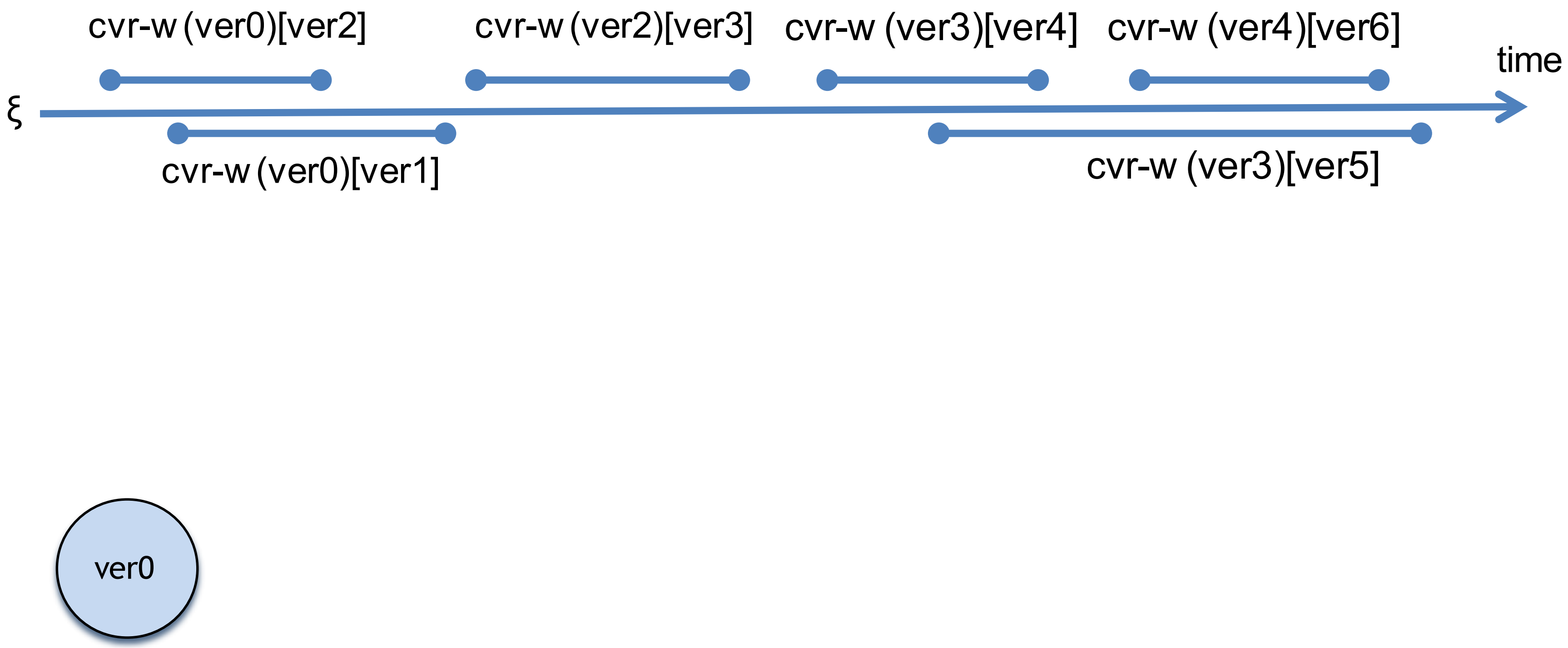
Evolution in Execution ξ

- If $\pi 1 = crv\text{-}w(ver_i)[ver_{i+1}]$ and $\pi 2 = crv\text{-}w(ver_j)[ver_{j+1}]$, two successful writes, *s.t.* $ver_i \leq ver_j$, then $ver_{i+1} < ver_{j+1}$



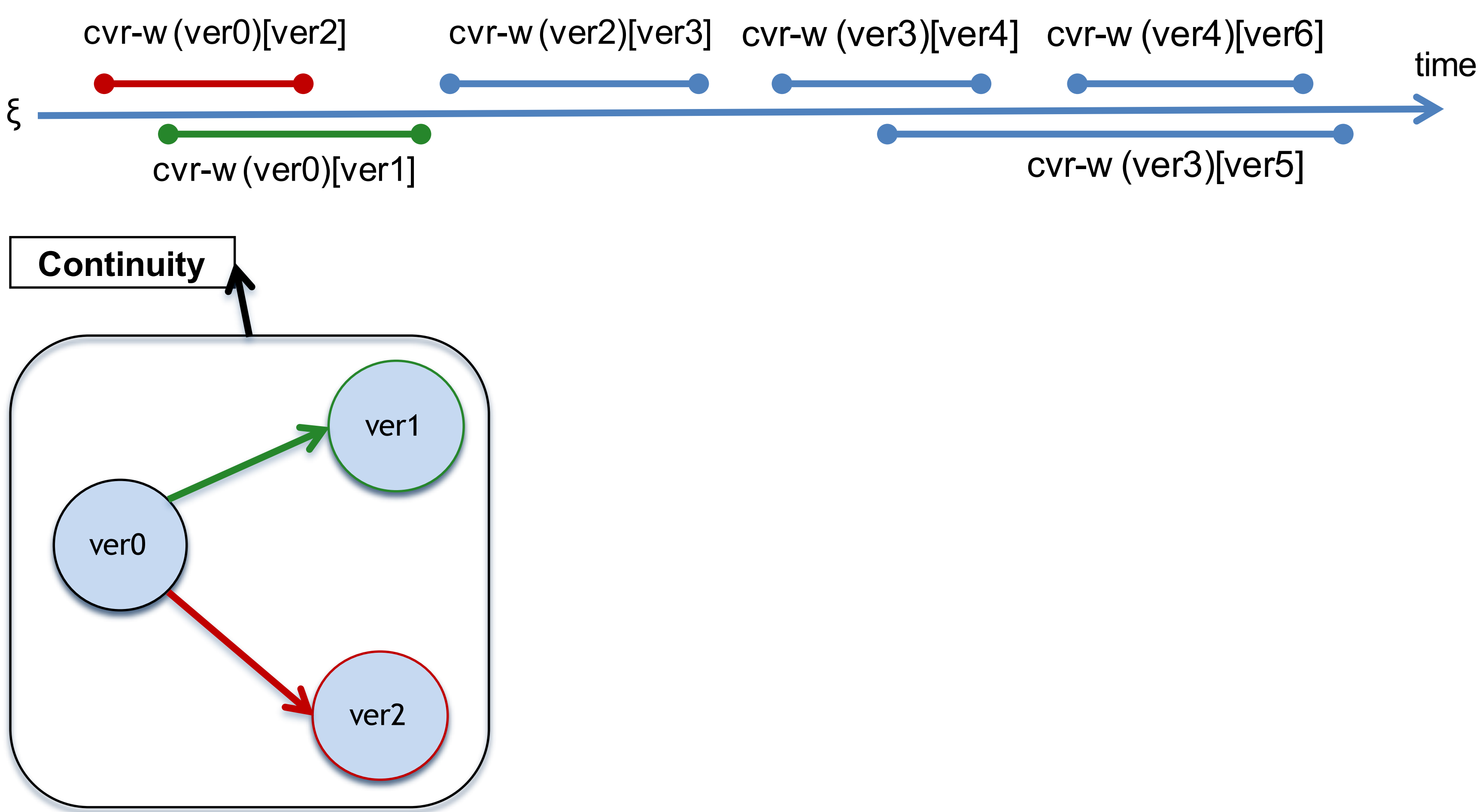
$$\boxed{ver_i < ver_k \textit{ AND } ver_z < ver_k}$$

Coverable Execution Tree



Total Order:
cvr-w (ver0)[ver1] $<_\xi$ cvr-w (ver0)[ver2] $<_\xi$ cvr-w (ver2)[ver3] $<_\xi$ cvr-w (ver3)[ver5] $<_\xi$ cvr-w (ver3)[ver4] $<_\xi$ cvr-w (ver4)[ver6]

Coverable Execution Tree

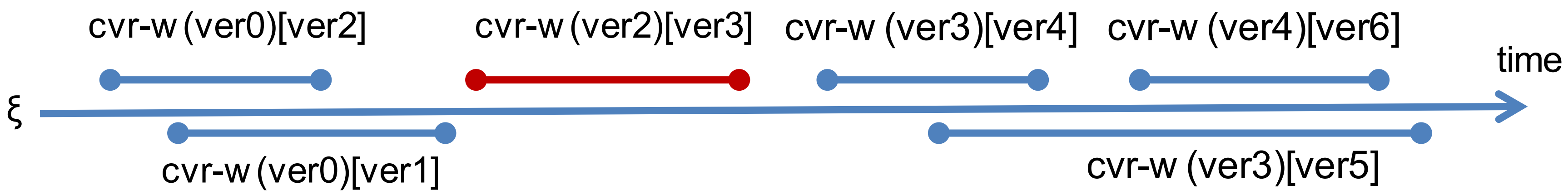


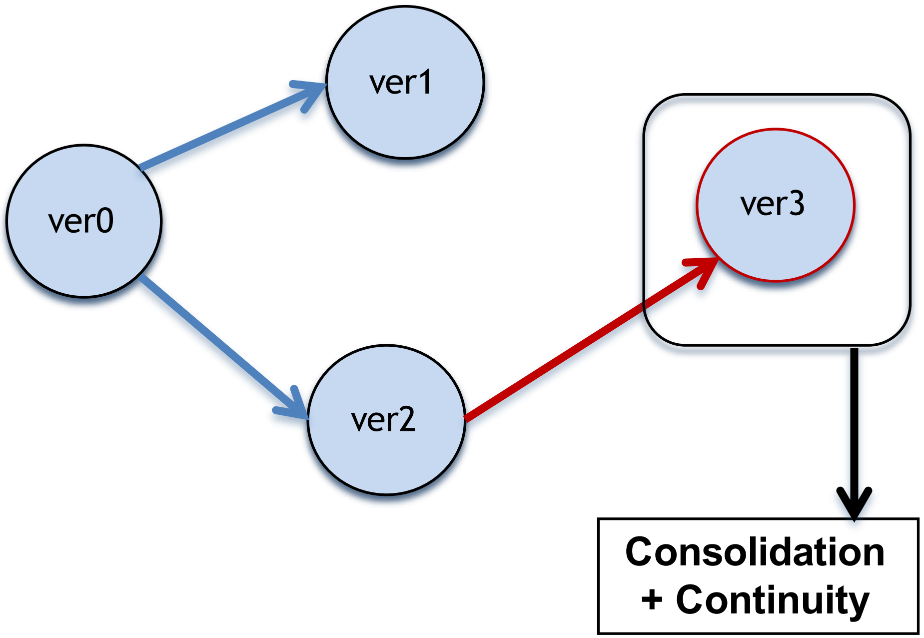
Total Order:
cvr-w (ver0)[ver1] $<_\xi$ cvr-w (ver0)[ver2] $<_\xi$ cvr-w (ver2)[ver3] $<_\xi$ cvr-w (ver3)[ver5] $<_\xi$ cvr-w (ver3)[ver4] $<_\xi$ cvr-w (ver4)[ver6]

1/17/17

Nicolas Nicolaou

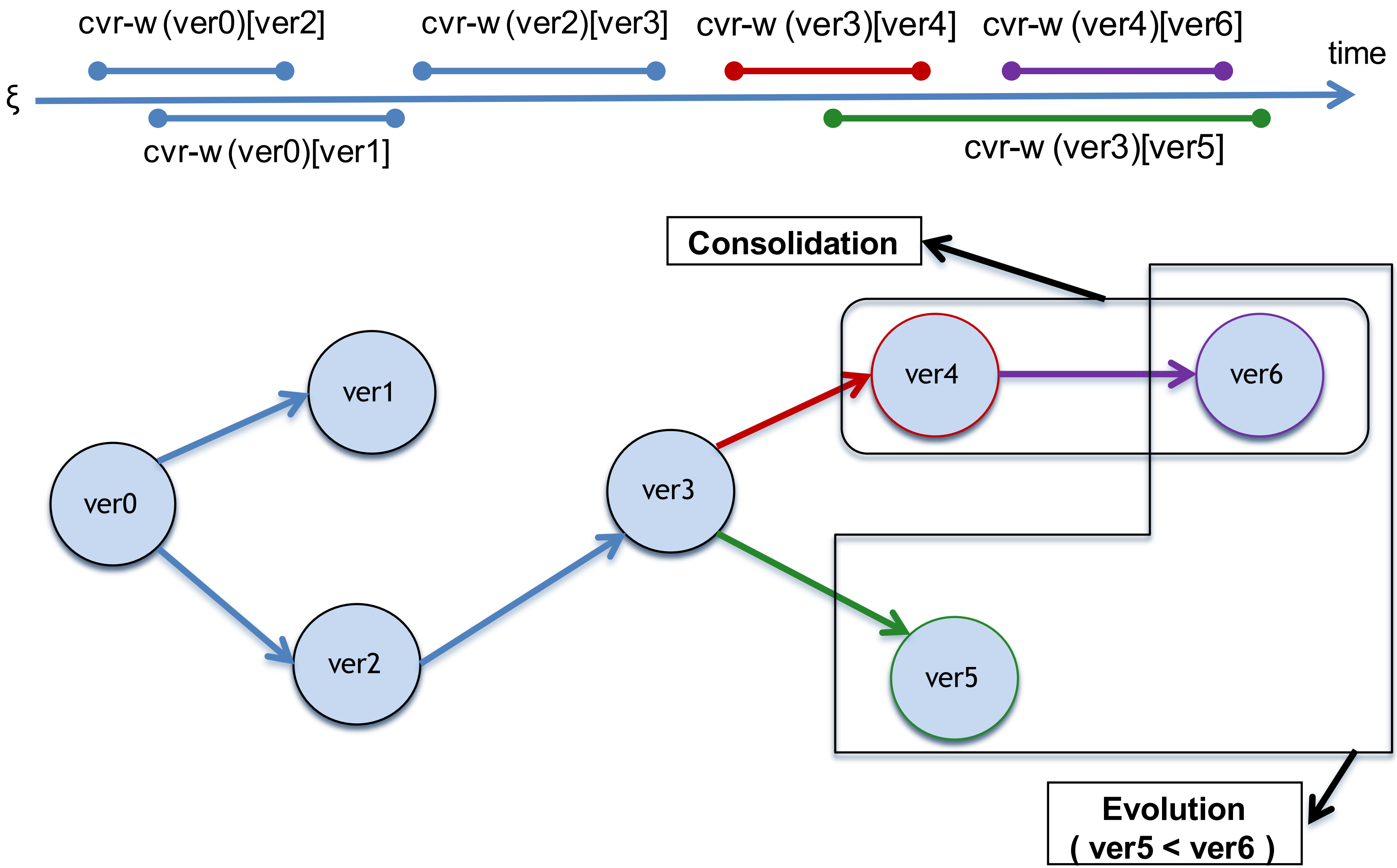
Coverable Execution Tree





Total Order:
cvr-w (ver0)[ver1] $<_{\xi}$ cvr-w (ver0)[ver2] $<_{\xi}$ cvr-w (ver2)[ver3] $<_{\xi}$ cvr-w (ver3)[ver5] $<_{\xi}$ cvr-w (ver3)[ver4] $<_{\xi}$ cvr-w (ver4)[ver6]

Coverable Execution Tree



Total Order:
cvr-w (ver0)[ver1] $<_\xi$ cvr-w (ver0)[ver2] $<_\xi$ cvr-w (ver2)[ver3] $<_\xi$ cvr-w (ver3)[ver5] $<_\xi$ cvr-w (ver3)[ver4] $<_\xi$ cvr-w (ver4)[ver6]

Definition: Coverable Atomic Register

- A versioned register is coverable and atomic, referred as coverable atomic register, if any execution on the register satisfies:
  - Atomicity, and
  - Coverability with respect to the total order imposed by atomicity on the successful writes.

Coverable vs Ranked Registers

Ranked Registers:

- Registers that support operations where value is associated with a rank
- Writes may commit or abort
- Writes abort => exists a rank higher than the one to be written
- May write any arbitrarily large rank

Theorem:

There is no algorithm that implements a coverable register using a set of ranked registers.

1/17/17
Nicolas Nicolaou

Implementing Coverable Atomic R/W Registers

- Enhance ABD algorithm [ABD96]
  - Register replicated to S servers
  - Use *<wid, ts, v>* triples to order operations
  - *tag = <wid, ts>* is used as the version of the object
  - tags are compared lexicographically

Server Protocol (Upon rcv a msg m from i)

```
• if m.type!= Query && m.tag > tag
    • Local <tag,v> = <m.tag,m.v>
• Send local <tag,v> to i
```

Implementing Coverable Atomic R/W Registers

cvr-write(val,ver) at w_i

- Phase 1:
  - Send Query to all
  - Collect <tag,v> from a majority
  - Discover max(<tag,v>)
- Phase 2:
  - If (ver == max(tag))
    - <tag',v'> = <<w_i,max(tag.ts)+1>,val>
    - flag = chg
  - Else
    - <tag',v'> = max(<tag,v>)
    - flag = unchg
  - Send Write max(<tag',v'>) to all
  - Collect ack from a majority and return (<tag',v'>,flag)

- Read is similar to the write operation except that in Phase 2 it propagates the max(<tag,v>) discovered.
- Read returns the max(<tag,v>) discovered during Phase 1

Application: Weak RMW

- Implement weaker RMW semantics using coverable R/W registers
  - Not all RMW operations may modify the value of the shared RMW object
  - In the case of concurrency at least one will RMW the object

weakRMW(F)

```
• <oldval,lcver> = cvr-read()
• newval = F(oldval)
• <lcval,lcver,flag> = cvr-write(newval, lcver)
• If (flag == chg)
    • return <lcval, success>
• Else
    • return <lcval, failure>
```

Nicolas Nicolaou

Application: Concurrent File Objects

- File operations: get(), revise()
- A get() operation returns the latest version of the file
- A revise() operation specifies the version of the file to be modified
  - Prevent revising older versions

Get()

```
• <lcval,lcver> = cvr-read()
• return <lcval, lcver>
```

Revise(v, ver)

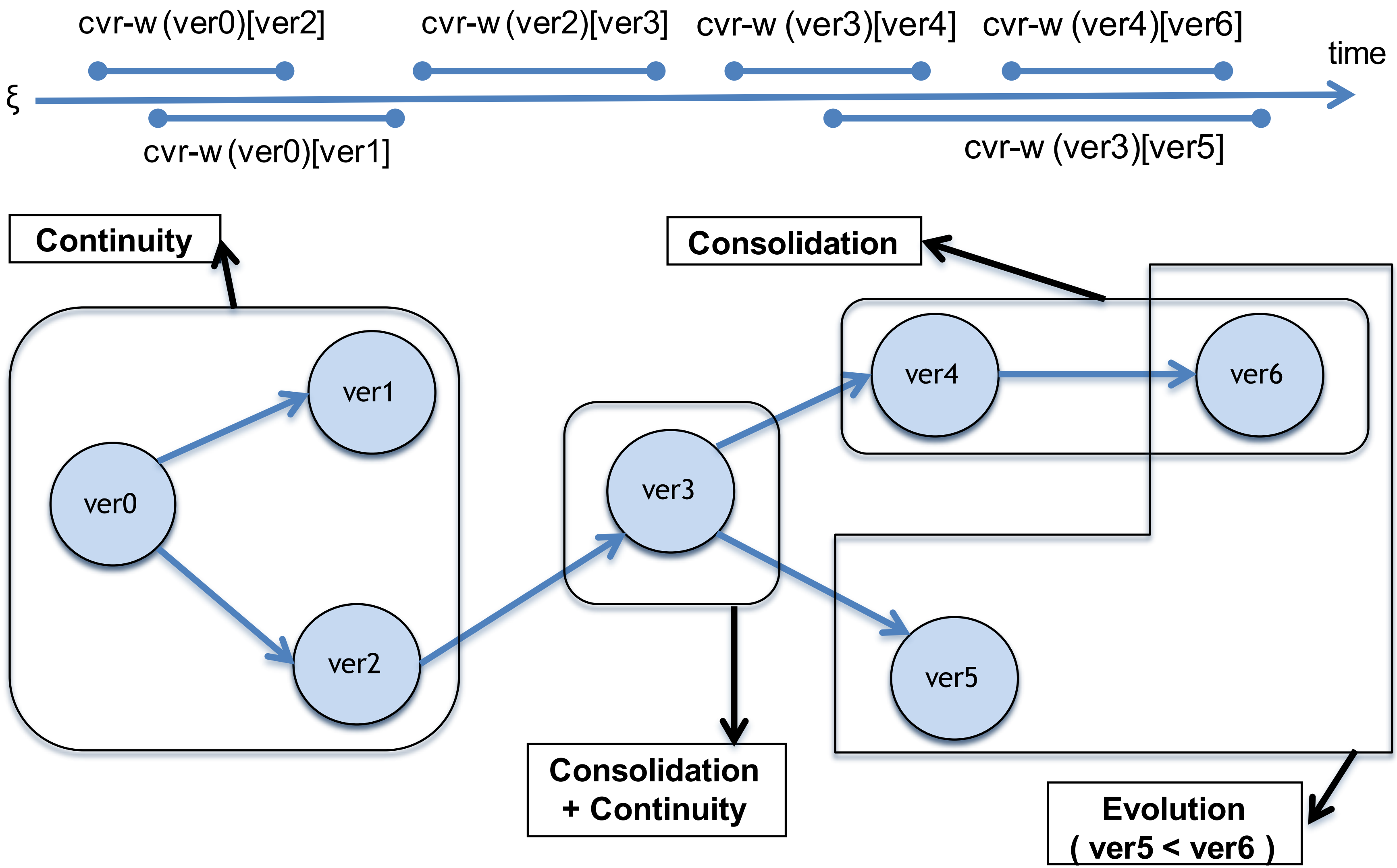
```
• <lcval, lcver, flag> =
  cvr-wite(v, ver)
• If (flag == chg)
    • return OK
• return <lcval, lcver>
```

Conclusions

- We defined a new object type: versioned object
- We defined Coverability – a new property for concurrent versioned objects
- We defined Atomic Coverable R/W Registers
- We compared ACR similar objects (Ranked Registers)
- Demonstrated an implementation of ACR
- We presented two applications
  - Weak RMW objects
  - Distributed File objects

Thank you !

Coverable Execution Tree



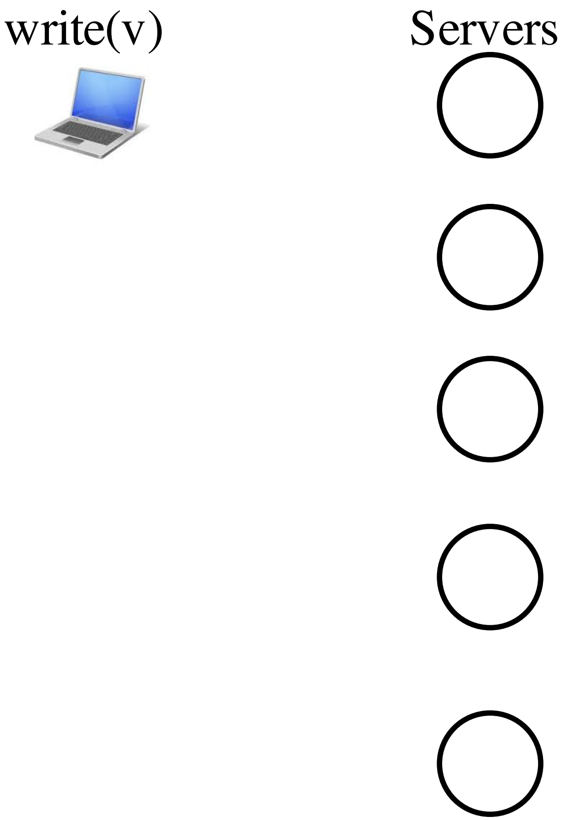
Total Order:
cvr-w (ver0)[ver1] $<_\xi$ cvr-w (ver0)[ver2] $<_\xi$ cvr-w (ver2)[ver3] $<_\xi$ cvr-w (ver3)[ver5] $<_\xi$ cvr-w (ver3)[ver4] $<_\xi$ cvr-w (ver4)[ver6]

Contributions

- Define Versioned Objects
- Define CoVerability: a new property for distributed, versioned, objects
- Define Atomic Coverable R/W Registers (ACR)
- Compare ACR with Ranked Registers
- Use ACR to implement
  - Weak RMW objects
  - Distributed File objects

Algorithm ABD: Recalling the past

- [Attiya, Bar-Noy, Dolev 1996] (Dijkstra Prize 2011)
- Order Operations by using <ts, v> pairs.



Writer Protocol

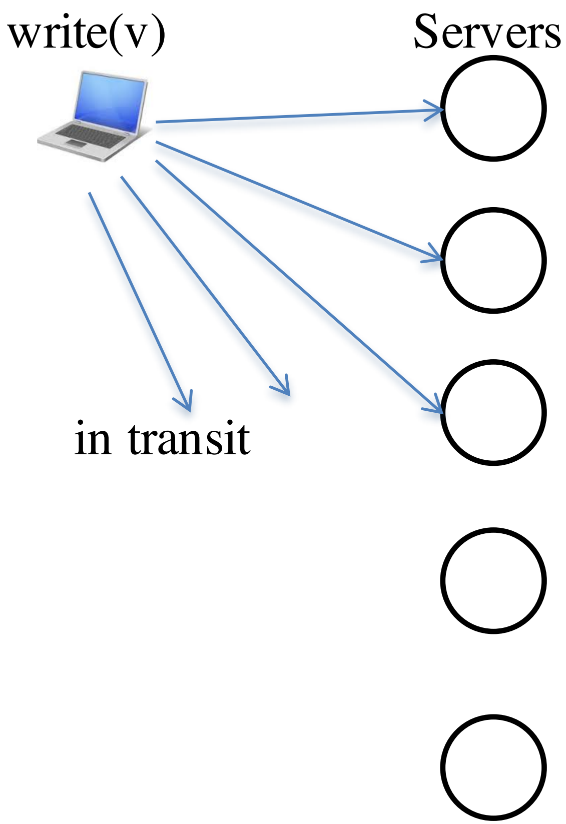
- ts++ //increment ts
- Send <ts,v> to all
- Wait for a majority to reply

Server Protocol (Upon rcv a msg m from i)

- if m.ts > ts
    - Update local <ts,v>
- Send local <ts,v> to i

Algorithm ABD: Recalling the past

- [Attiya, Bar-Noy, Dolev 1996] (Dijkstra Prize 2011)
- Order Operations by using <ts, v> pairs.

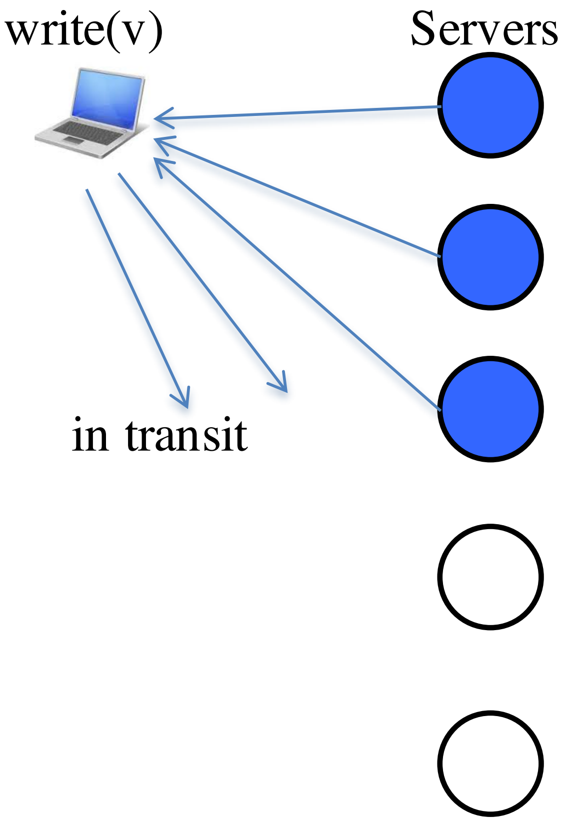


Writer Protocol

- `ts++ //increment ts`
- `Send <ts,v> to all`
- `Wait for a majority to reply`

Algorithm ABD: Recalling the past

- [Attiya, Bar-Noy, Dolev 1996] (Dijkstra Prize 2011)
- Order Operations by using <ts, v> pairs.

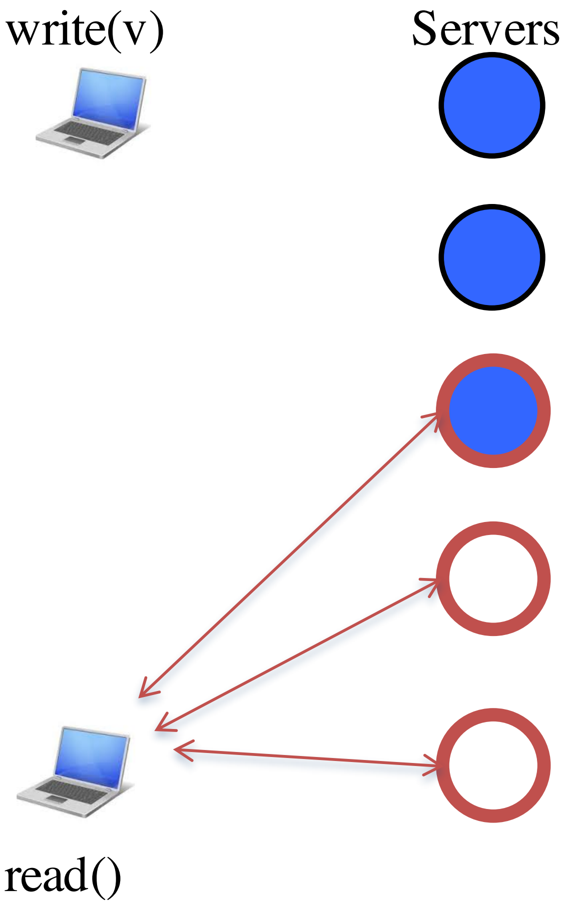


Writer Protocol

- `ts++ //increment ts`
- `Send <ts,v> to all`
- `Wait for a majority to reply`

Algorithm ABD: Recalling the past

- [Attiya, Bar-Noy, Dolev 1996] (Dijkstra Prize 2011)
- Order Operations by using <ts, v> pairs.



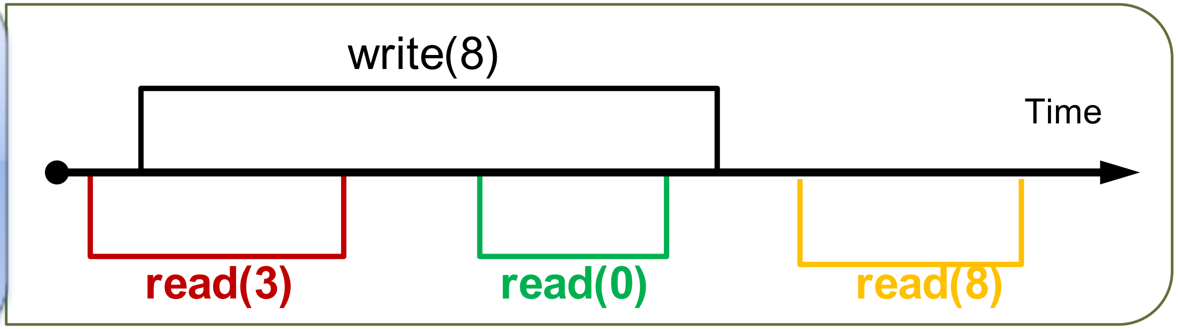
Reader Protocol (2 phases)

- Phase 1:
  - Send read to all
  - Collect <ts,v> from a majority
  - Discover max(<ts,v>)
- Phase 2:
  - Send max(<ts,v>) to all
  - Collect ack from a majority and return v

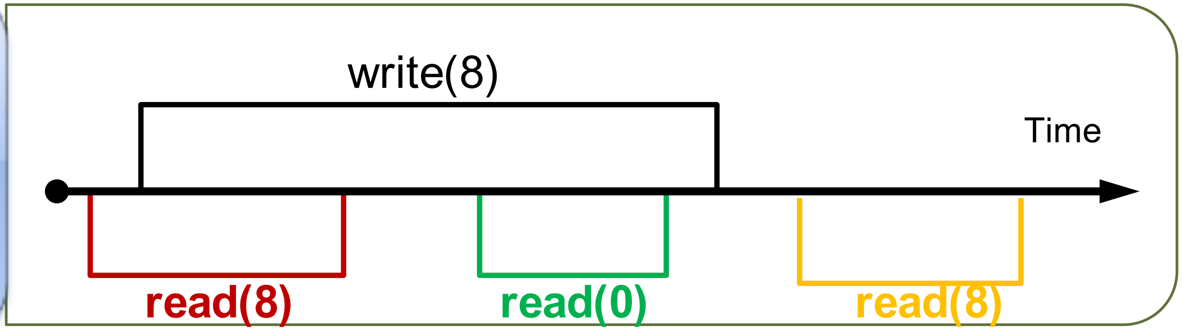
Reads must Write!

Consistency Semantics [Lamport86]

Safety



Regularity



Atomicity