A Survey of Distributed Garbage Collection Techniques

David Plainfossé, Marc Shapiro

- 1 -
INTRODUCTION

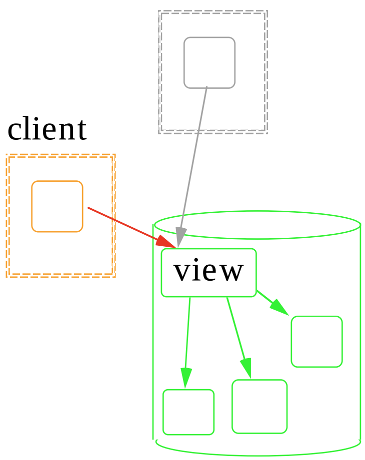
Example: database query
References in distributed systems and GC
State of the art: extensions of centralized algorithms, hybrid algorithms, distributed shared store algorithms
Definitions

Example : database query

Query returns reference to view
Use reference in future queries
Pass to other clients

- Keep how long?
- Pass to other clients?
- What happens when clients finish?

Persistence By Reachability



References in distributed systems

Sound
- Referential integrity

Efficient, large scale
- Avoid layering
- Avoid extra messages, synchronization, etc.
- At odds with complete GC

Usable
- Tolerate faults
- Sharing, caching, replication, etc.

GC difficult, new

Definitions

References: directed graph between objects

Root: distinguished uncollectable object

Reachable object: ∃ reference path from root

Mutator: application code, modifies graph

- create objects
- assign references
- cause objects to become unreachable: **garbage**

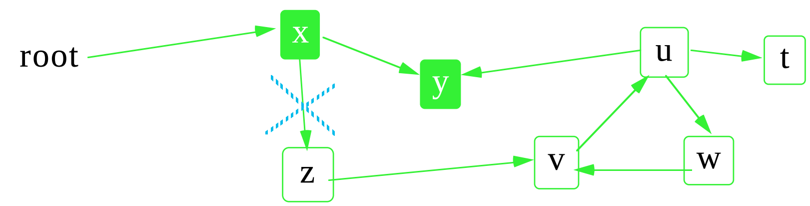
Collector: garbage collection system

- detects unreachable objects
- reclaims space

Garbage collection

Garbage = unreachable from root

Global, stable property



Counting algorithm:

Count handles to object

Local: incomplete (cycles)

Tracing algorithm:

Walk graph from root

Objects not visited are garbage

Global: doesn't scale

State of the Art (1) Centralized algorithms

Centralized GC:

- counting: incomplete
- tracing: in-place or compacting

Multiprocessor tracing:

- global termination
- consistent memory
- **barrier**: mutator-collector synchronization

State of the Art (2) Extensions of centralized algorithms

General algorithms:

- snapshop
- causal ordering
- transaction

Correctness guaranteed

Too strong

Distributed GC:

- Counting: scales, incomplete, non-FT
- Ladin & Liskov: centralized
- Hughes: periodic termination

State of the Art (3)
Hybrid algorithms

Per-space tracing + inter-space counting:
- scalable (independent)
- ¬ complete

SGP algorithm + SSP Chains:
- fault-tolerant
- performance
- scalable

Broadcast

Lang-Queinnec-Piquer:

Trace groups ⇒ more complete

Dynamic groups

Very complex, high cost

State of the Art (4)
Distributed Shared Store

Shared store ⇒ caching + replication

DSM: worst-case model

Communication, consistency: high cost

Larchant
- tolerates inconsistency
- replicated tracing in cache

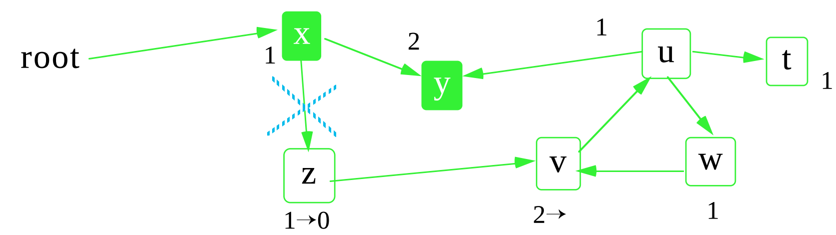
Broadcast

- 2 -
COUNTING ALGORITHMS

Naïve distributed reference counting

Weighted Reference Counting (WRC)

Reference counting



Invariant: count == #refs

Continuous; cost

Cycles of garbage are not reclaimed

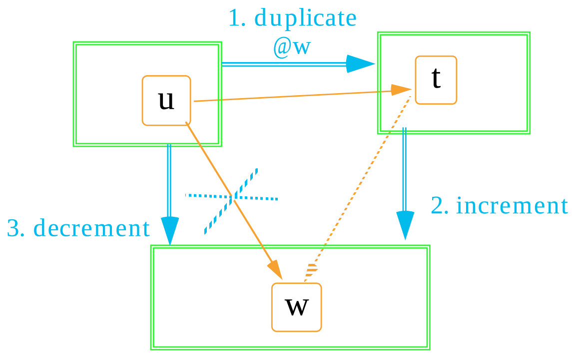
Local: scales easily

Create object:
- counter := 1
- return reference

Duplicate reference:
- counter += 1

Delete reference:
- counter –= 1
- if counter == 0 then reclaim

Problems of naïve distributed RC



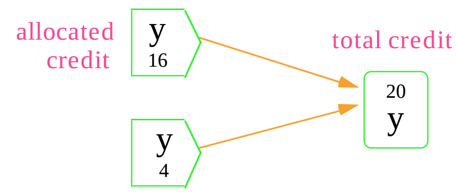
Costly

Doesn't tolerate message faults

loss; out-of-order

Strong invariant

Weighted Reference Count



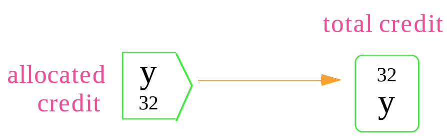
Invariant : $\sum ac = tc$

Increment distributed ➠
message order
unimportant

Reliable messages:
strong invariant

Incomplete

WRC create

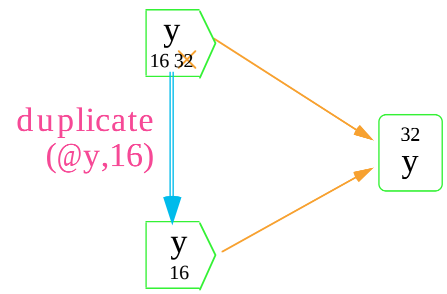


Initial credit: a power of 2

Allocated credit: exponent only

Total credit: number

WRC duplicate

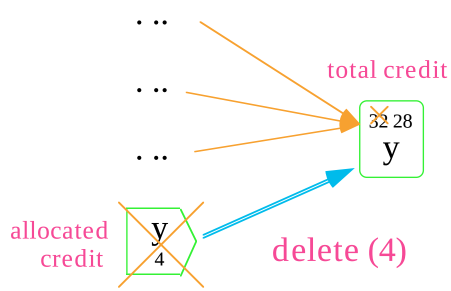


Pass on half of allocated credit
(subtract 1 from exponent)

Application message only; reliable

No communication with target

WRC delete



Reliable message to target

Subtract allocated credit from total credit

If goes to 0, deallocate

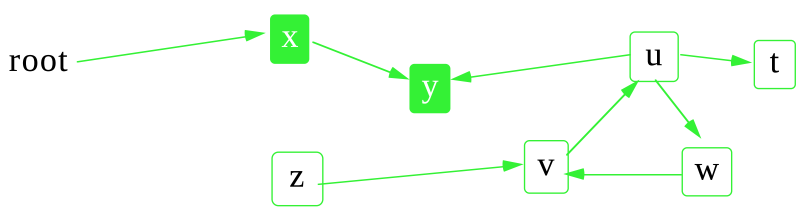
- 3 -
TRACING ALGORITHMS

Naïve distributed Mark and sweep

Hughes

Ladin & Liskov

Mark and sweep



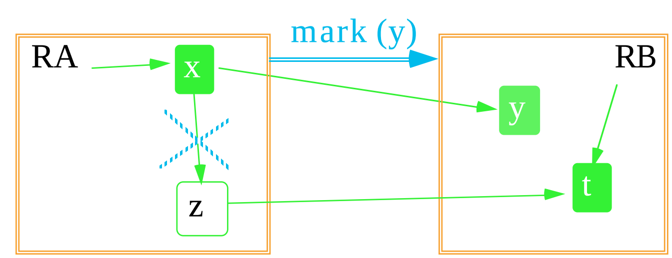
Collector:

1. mark: walk from root
2. sweep (linear scan): if not marked, reclaim

Complete

Beneficial side effects

Naïve distributed M&S



1. Mark in parallel

   Remote: propagate to target site
2. Sweep in parallel

Fault tolerance: abort & restart

2 global barriers: won't scale

Hughes' algorithm (1)

Global clock

Reachable ⇒ timestamp increases

Timestamp < global minimum ⇒ unreachable (sweep)

Gobal minimum: global algorithm

Effect: multiple parallel M&Ss

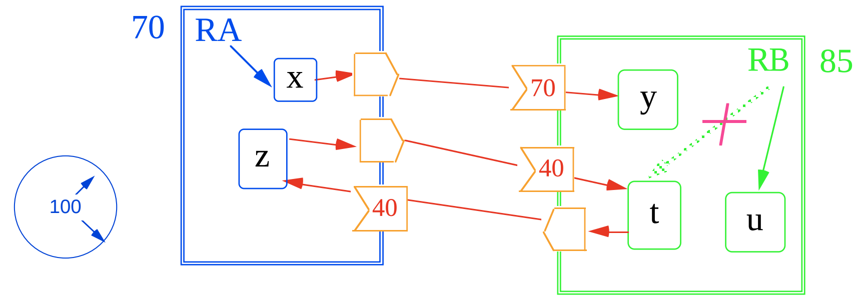
Local GC: repeat:

- timestamp root with current time
- propagate copy father's timestamp to son
- higher timestamp takes precedence

Sweep in parallel: repeat:

- if timestamp<minimum then reclaim

Hughes' algorithm (2)



B40	mark: z:=40		*B85*	mark. z:=40
B50	destroy RB →		*100*	global min=70
A70	mark: y:=70 t:=40		*A115*	sweep: reclaim a

Ladin & Liskov algorithm (1)

Trace from

local root ∪ {in-refs}

Send **paths** to service

Central service simulates (approximate) graph, traces, tells spaces of unreachability

Ladin & Liskov algorithm (2)

Tolerate relativistic effects:

- timestamps
- conservative

Tolerate incomplete information:

- conservative ➠
- completeness ➠ Hughes'

within service: no global termination

Tolerate failures:

- reliable messages
- stable store
- replicate service

- 4 -
HYBRID ALGORITHMS

Hybrid: trace + count
Lang, Queinnec, Piquer
SSP Chains + SGP: fault tolerance, races

Hybrid algorithms

Within space: trace

- complete
- fault tolerant

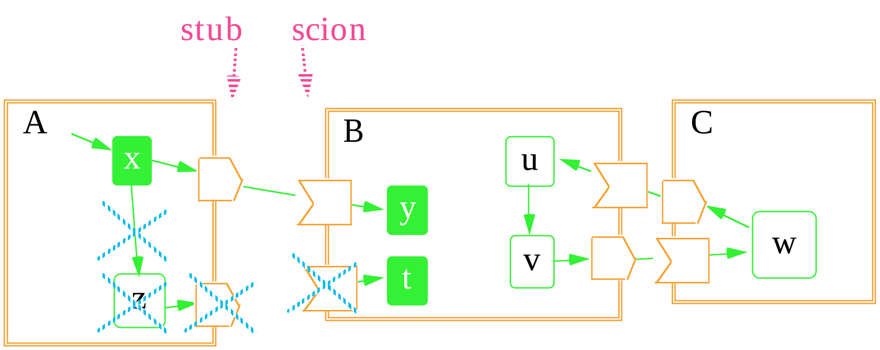
At space boundary: reference count

- scale

Issues:

- fault tolerance
- relativistic effect
- garbage cycles

Hybrid: (1) trace within each space

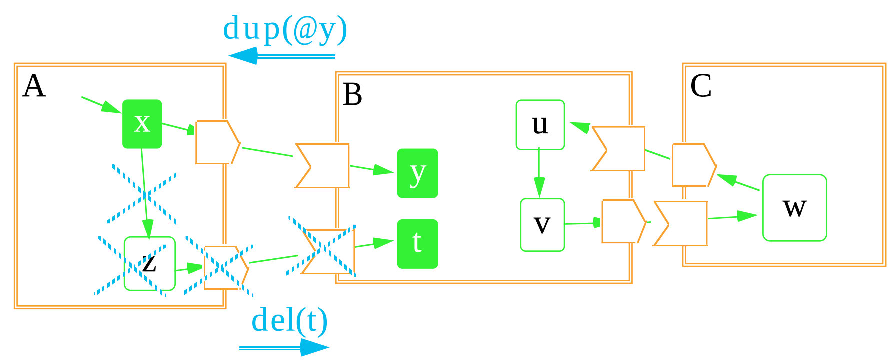


Local GC: per-space GC (SGC)

- Roots = local root ∪ {scions}
- Trace independently of other spaces

No synchronization

Hybrid: (2) count between spaces



Export reference: **scion++**

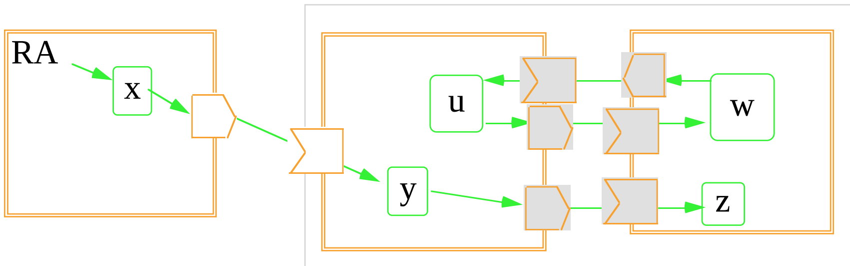
user message

Stub unreachable: **scion—**

control message

Garbage cycles not collected

Lang, Queinnec, Piquer: garbage cycles



Form **group** dynamically

Christopher's algorithm

Disband: restore RCs

Chistopher:

- M&S from scions at group boundary
- temporarily adjust RCs of scions reached
- if internal scion has non-zero RC, reclaim

Lang, Queinnec, Piquer: properties

Global synchronization: ¬scale

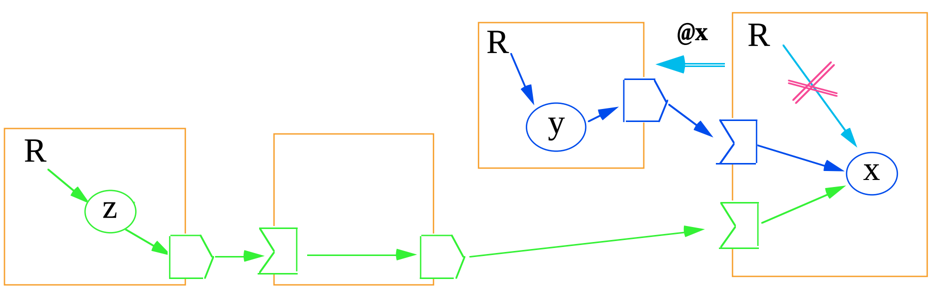
- create group
- terminate mark phase
- terminate sweep, disband

Reclaim garbage cycles

- **within** group
- all cycles: hierarchy of groups

Concurrent groups, M&Ss: extra complexity

SSP Chains & SGP Algorithm



Reference listing

Send: create scion

Receive: create stub

Invoke: shortcut

Optimizations

Within space: trace

Between spaces:

fault-tolerant RC

Fault tolerance

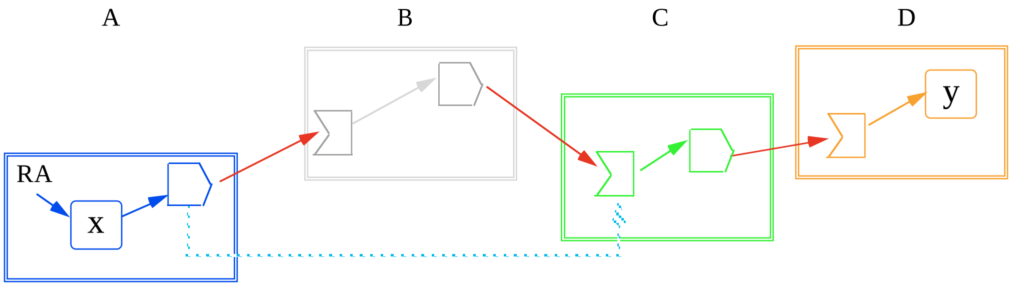
Issues:

- mutator message lost
- control message lost
- message races
- crash

Safe inconsistencies allowed

- duplicate reference: local, conservative
- **live** message: idempotent, sent multiple times
- timestamps
- recovery protocol

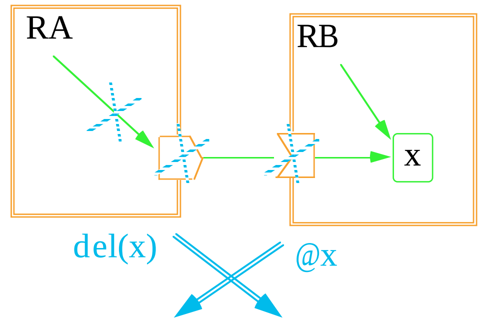
Crashes and recovery



C crashes:

- recovers: wait
  only directly involved objects uncollectable
- terminates: **mend chains, collect**

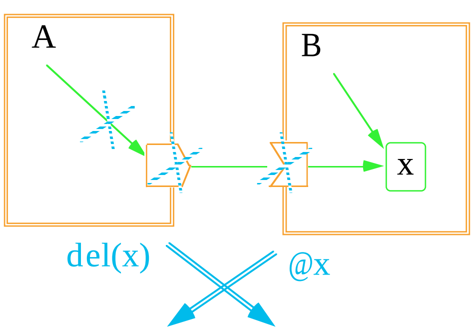
Create/delete race (1)



No global ordering: race between create and delete message

1. x reachable from A, B
2. (Picture)
   A sends @x to B (again)
   B deletes @x, sends del(x)
3. B receives @x, creates stub **without scion!**

Create/delete race (2)



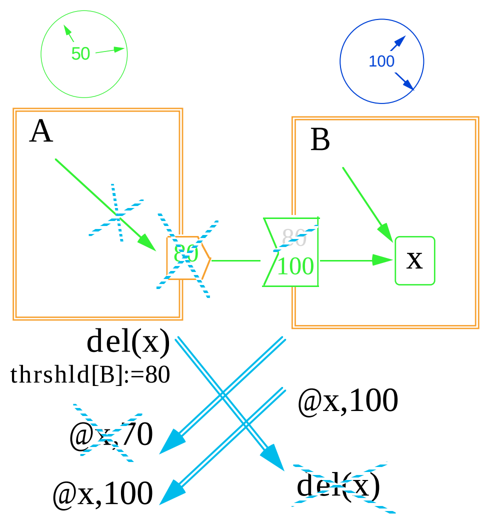
The problem: **no record of past, causality**

Solutions:

- causal protocol (complex)
- atomic protocol (costly)
- reference counts (not fault tolerant)
- detect race

➠ **timestamp**

Using timestamps to avoid races



Local timestamps

Scions, messages, stubs, delete: timestamped by sender of reference

Conservative:

- remove scion only if no message in transit
- drop message possibly creating stub with no scion

- 5 -
COLLECTING A DISTRIBUTED SHARED STORE

Limitations of classical model
Persistence By Reachability
Issues: consistency, scale, cost
Larchant

Limitations of classical distributed system model

Classical system model:
communication by messages

- No shared memory
- No replication
- No persistence
- No caching
- No groups

Unlike advanced systems

Persistence By Reachability ⇒ tracing GC

Long-term sharing: **persistent data**
A points to B:

A persistent ⇒ B persistent

Trace from **persistent root**
Multiple roots, garbage cycles: reference counting not adequate
Other benefits: **locality**, correctness, programmer productivity

GC issues in a shared persistent store

Scale
⇒
Cost of I/O prohibitive
⇒
Most objects live
⇒
Mutator performance
⇒
Replication, caching
not coherent
Complete GC not primary objective

GC consistency: easier than general consistency

Mutator:
- duplicate

Coherence:
- propagate
- ownership

Collector:
- create
- delete
- scan
- move, patch

References mutually independent

Not all interleavings unsafe

When in doubt, add to root *(⇔ don't collect)*

⇒ completeness problem

Larchant

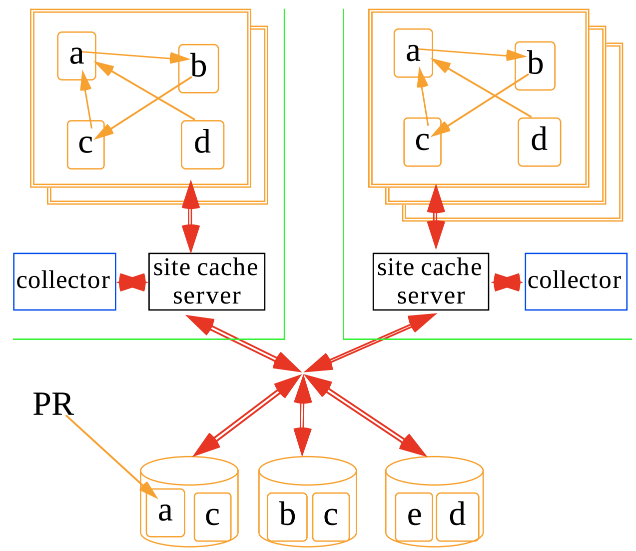
C/C++

Pointers

Single object universe

Reachability from Persistent Root

Incoherent memory



Larchant GC: main ideas

Trace from persistent root

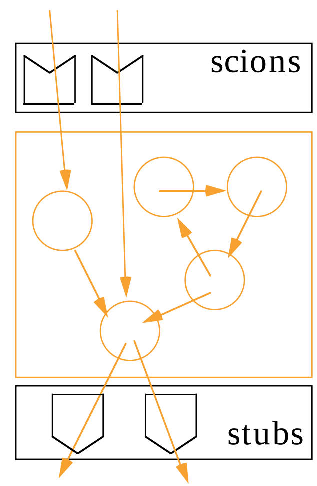
No global synchronization

Copying collector

Avoid I/O, messages, competing with application

- Partial, local **group** collections
- Dynamic, **opportunistic** groups
- GC uses **cached** data and locks
- Checked-out and unmapped clusters not collected
- Non-coherent: **union**

Collecting a single copy of a single cluster



GC:
- trace from scions
- create new stubs

Correctness: trivial

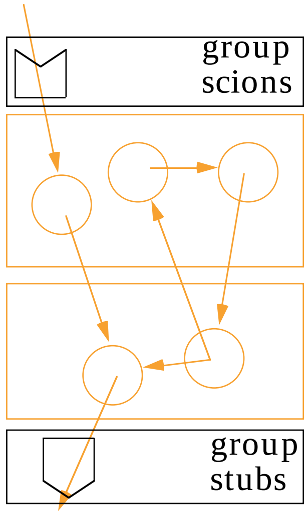
Conservative w.r.t. other clusters: incomplete w.r.t. dead cycles across clusters

No concurrent updates: no read or write barrier

Create not noticed until GC

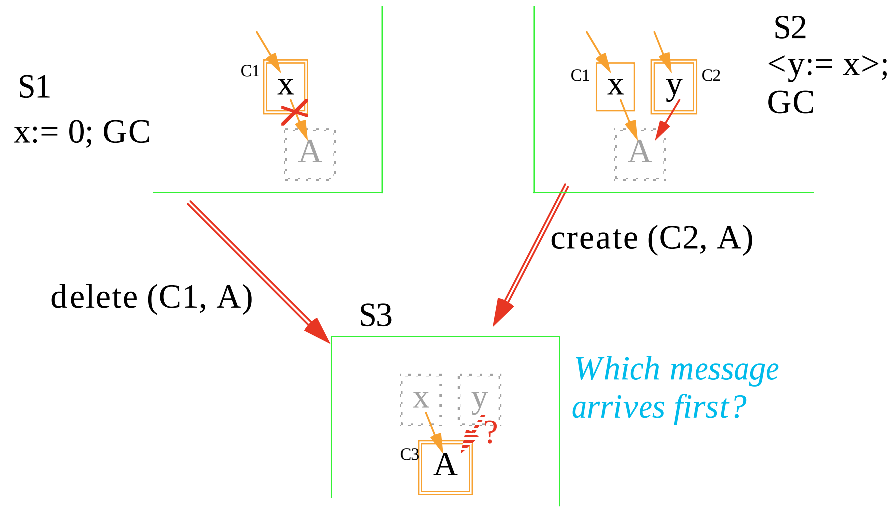
Client writes: abort GC

Collecting a group of clusters at a single site



- Complete w.r.t. clusters in the group
- Conservative w.r.t. clusters not in the group
- Locality-based heuristics
  - ➔ **segregate data sets:**
  - scan cached clusters
  - exclude checked out clusters

GC of a replicated cluster



Distributed store GC safety rules

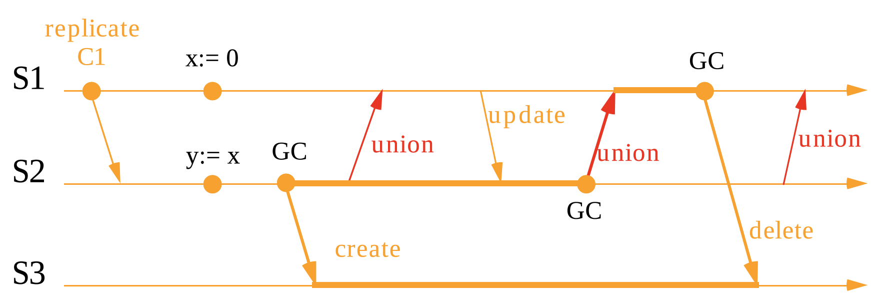
<u>x</u> points to <u>A</u>:
- in most recent version of <u>x</u>
- in incoherent version of <u>x</u>

Union rule: <u>y</u> live if reachable in any current version of <u>x</u>

Promptness rule: send <u>creates</u> before <u>deletes</u>

Causal delivery rule: <u>creates</u> and <u>deletes</u> delivered in causal order w.r.t. <u>union</u> messages

Asynchronous implementation of GC safety rules



Union rule

Causal delivery rule

- 6 -
CONCLUSIONS

Distributed GC: important practical & theoretical problem

Issues: consistency, asynchrony, fault tolerance, new performance issues

Safety: do not collect ⇒ **completeness** problem

SSPC:

- fault-tolerance ⇒ recovery
- cycles not solved

Larchant: partial collection ⇒ heuristics