The Emerald Spring Cleaning Garbage Collector

An Example of a Distributed, Robust Systems Application

Eric Jul

Professor, IfI, UiO

What’s my point?

A distributed “application” that is robust, decentralized, and works in a hostile environment.

Hostile:

- Machine crash (fail-stop) quite often.
- Non-crashed machines continuously modify distributed state.

What's the Problem?

Distributed Garbage Collection

Why interesting?

- Tough correctness criteria
- Exemplifies lots of interesting distributed system principles

Specific Problem

Take: A distributed OO system with object mobility

Already has:

- Fast, local GC for node-local garbage
- Non-comprehensive, non-robust Distributed GC

WANT:

Comprehensive, robust, on-the-fly collector

Criteria for Our Solution

Comprehensive: 100% of garbage collected

On-the-fly: System keeps running

Robust (despite wild Kalashnikov):

- Starts, runs, complete even when:
- N nodes booted up, run for some time.
- Kill 25% of nodes.
- Every 5 seconds: kill one more; reboot one!

Background

Emerald Virtual Machine

- (Original) compiler generates native code
- More or less std. virtual machine implemented as a single UNIX process
- All objects in memory in large, shared heap
- Software fault mechanism for calls to remote objects
- Remote Object Reference Table
  - an entry for each incoming/outgoing ref

High level view of problem

- Standard graph formulation:
  - graph nodes are objects
  - graph arcs are references
  - graph partitioned into non-overlapping parts; one for each location
  - each object is located at most at one node
  - immutable objects are omnipresent

Problem formulation

Build a distributed GC that starting from root objects will:

- remove all garbage objects: *comprehensive*
- operate while mutators continue: *on-the-fly*
- start, run, complete despite crashed machines: *robust*
- no single controller: *decentralized*

Let’s do it

Comprehensive

Algorithm: standard Mark and Sweep collector

Liveness: reachbility from root objects

Trace from root objects thru all reachable objects by scanning each live object for references

New objects are considered live by definition

Threads: *just consider activation records to be live "objects" in the graph.*

How on-the-fly?
An analogy:
Louis XIV at Versailles

King must experience a clean castle every morning.

Crew of thousand working early every morning *(stop-mark-and-sweep).*

Expensive…

New, cheap cleaning contract

- Promise king clean castle
- Use small, agile crew

How the Scam Works

- Clean the kings bedroom quitely as he sleeps (cheap, just one room)
- King wakes up: claim *entire* palace is clean!
- Invite king for inspection tour
- IF he moves toward another room:
  - stall him for a moment (bribe his court jester!)
  - quickly clean the room
  - let him enter the room

Liveness by Reachability

To be live, means that something live can REACH you.

(Something live might be yourself, e.g., if you have an executing process inside yourself.)

Transitive closure of reachable.

Objects as an Object Graph

For the purpose of garbage collection, the sea of objects is considered a graph:

- Each node is an object
- Each reference is an arc

Roots:

- Each process that can execute – or is waiting for, e.g., I/O, a timer, etc.
- Each object that is inherently rechable.

Classic MARK-AND-SWEEP

Start at the “roots” and simply trace thru the live part of the object graph.

As do this one step at a time, we add a marking to each object.

Classic black, gray, white marking

Black

object found to be live and object has been scanned and point to black/gray objects only.

Gray

object found to be live but the object has not been scanned and so may contain pointers to white objects

White

object liveness unknown; not scanned

Initially all objects are white *

Classic Mark and Sweep

Classic Mark and Sweep:

- mark each of the roots grey
- for each gray object:
  - mark each referenced white object gray
  - thereafter mark the object black
- sweep thru entire storage, deallocating any white object: it is garbage!

*

Our On-the-fly Operation

Want Collector to work while Threads (mutators) continue executing:

- Stop all threads on ready queue
- Repeatedly
  - pick one
  - scan and mark top activation record black
  - put it back on the ready queue
- Scan and mark all roots black

Important Invariants

1. Black objects only point to black or gray
2. Black or gray objects are live
3. Mutators only see black objects
4. New objects are created black
5. Once black always black
6. Once gray eventually black

Object Faulting

On invocation of a gray object:

- fault – uses the software fault designed for catching remote invocations
- scan object refs and mark refs gray
- mark object black
- continue invocation

An example of a *Read Barrier*

Background collector

Keep a set of gray objects.

Let the collector run a background thread that repeatedly takes a gray object, and makes it black – by scanning it and marking referenced objects gray (if not black already).

Multiple Machines

- Any random machine starts a Spring Cleaning Collection
- ANY interaction with another machine includes a piggybacked notice to start a new collection.
- A background collector on each machine.

Remote Invocations

- Source object is black, destination may be gray or white
- On arrival:
  - start the (node-local) collection, if not started
  - scan the destination object marking its refs gray
  - mark the destination object black

*

Scanning Remote Refs

During a scan we may meet a reference to a remote white object.

Mark the OUTGOING reference gray in the object table

Eventually send a “mark gray” message to the machine that holds the object which then makes it black – and returns a “made black” message.

Batch multiple “mark gray” requests.

Same for “made black” requests.

Piggyback on any net-traffic – or eventually send.

Checkpointed Objects

Objects may checkpoint themselves.

Non-checkpointed objects just go away on crash.

Checkpointed ones come back – with a lot of old references!

Recovery section can reinitialize the object, e.g., check that references are still valid and fix invalid references.

Robustness

Our Spring Cleaner must periodically checkpoint its state.

When node reboots:

- Restart the collection from checkpoint
- mark references to dead objects black

Crashed Machines

- Assumption: eventually a crashed machine reboots (if permanently down: easy!)
- GC State of objects also checkpointed.
- A collector with a gray ref. to a downed machine must wait for the downed machine to reboot.
- Eventually the gray ref. is “pushed” to the other machine – and eventually a “made black” is returned.
- If the gray ref is to a non-checkpointed object: forget it.

Termination Problem

Collection ends when every gray set on every machine is empty SIMULTANEOUSLY

Non-trivial to detect

What’s the problem?

When to start detecting termination?

How to do it?

Termination Solution

Solution is well-known: Distributed Consensus

Our solution uses a two-phase commit termination protocol: any report of a gray object nullifies the termination attempt

ANY machine can initiate the termination attempt – and any one that detects the termination commit can declare the mark phase done

Fast, local Garbage Collector

- Spring Cleaning Collector: slow.
- Have a node local collector, e.g., a node local version that runs frequently and fast.
- Each remotely referenced object is marked as externally visible (one bit) when a reference to it is given out to another node.
- Local collector considers ALL such marked objects as LIVE – puts them in the root set.
- Local collector has its *own* set of marking bits.

Sweep Phase

- Simple: go thru Object Table and reclaim objects marked white.
- UPS: Dual Collector Design: risk both local and Spring Cleaning Collector tries to reclaim same object!

Spring Cleaning Sweep Phase

- Each remotely referenced object is marked as externally visible (one bit)
- Sweep merely resets this bit(!)
- Node-local collector actually does the reclaiming
- Advantage: no synchronization conflicts with local collector

Mobility Complications

Remote object reference system uses forwarding addresses.

Forwarding address chains may be broken by crashed machines.

Broken Forwarding address chains are reliably fixed.

Forwarding addresses are colored.

Objects in transit: must be scanned before transit.

Pipelining

- Can be pipelined
- Multiple coloring bits: we use 4x, so up to three collections outstanding.
- Later collections help older collections (if live later, certainly live earlier).
- Sequence number: anyone can bump it and start a new collection within the 4x window.

Spring Cleaning

- Run seldom
- Slow: may have to wait days for downed machines to reboot
- Permanently dead nodes manually declared dead

Conclusion

Non-trivial, distributed application operating in a hostile “Kalashnikov” environment

URLs

See Teaching Material:

Two papers: one workshop paper and Niels Chr. Juul’s Ph.D.

This PowerPoint presentation

OOPSLA 2000 GC Tutorial