How to Write a Distributed Garbage Collector

Name: Ari Zilka, Saravanan Subbiah
Terracotta, Inc.
Goals

- Learn a little...
- about Terracotta
- about garbage collection
- about distributed algorithms
- about complexity
Agenda

- Hypothesis
- Case study: virtual heaps / distributed gc
- Terminology
- Step 1: DGC in single Terracotta server
- Step 2: Adding generational collection
- Step 3: DGC for Terracotta server arrays
- Conclusions
Agenda

> Hypothesis
> Case study: virtual heaps / distributed gc
> Terminology
> Step 1: DGC in single Terracotta server
> Step 2: Adding generational collection
> Step 3: DGC for Terracotta server arrays
> Conclusions
Hypothesis
Algorithm design outweighs platform importance

> Some say, “don’t distribute” but the push is in the opposite direction

> distribution / multi-threading is more about algorithms than platform
  • Asynchronous nature leaks everywhere
  • Variations in environment and inputs force this leak
  • the more “bare metal”, the better ?!?!
Agenda

> Hypothesis
> Case study: virtual heaps / distributed gc
> Terminology
> Step 1: DGC in single Terracotta server
> Step 2: Adding generational collection
> Step 3: DGC for Terracotta server arrays
> Conclusions
Case Study: Virtual heaps
What are they?

> Terracotta can make application heap page in and out of the JVM machine
> It can persist application heap to disk
> It can provide centralized object identity across distributed JVMs across a TCP/IP network
Virtual heap pictorial
Virtual heap pictorial
Virtual heap pictorial
Virtual heap pictorial
Virtual heap pictorial
Virtual heap pictorial

Java Virtual Machine
Application instance

Terracotta Server
app heap in memory

page to Terracotta

app heap on disk

page to disk

Terracotta Server's disk

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
The Garbage Collection Challenge
Virtual heaps represent a distributed computing challenge

> How do we collect garbage in a distributed world?
> No one layer has “everything”
> Crawling disk on Terracotta server is too slow
> Crawling memory in application instances is incorrect as heap is now virtual

> NOTE: platform would not help with any of these issues...
Agenda

> Hypothesis
> Case study: virtual heaps / distributed gc
> Terminology
> Step 1: DGC in single Terracotta server
> Step 2: Adding generational collection
> Step 3: DGC for Terracotta server arrays
> Conclusions
Terminology
Let’s get on the same page

> **L1** application node, JVM machine
> **L2** Terracotta server instance
> **root** top-most object in a graph clustered by Terracotta (for example, “a”)

> garbage in Terracotta → garbage in Java
  • no reference from any object reachable from roots
  • not resident in any application heap
How distributed garbage gets created
the last bit of context...

> Create a shared object by adding it to a shared graph
> Remove the shared object from the graph in L1
> GC in L1 collects object
> Now the object is garbage and needs to be collected
> Collect it in the L2
> Example: map.put( k,v); map.remove(k); // value is garbage
Creating distributed garbage
An existing object graph

// #1 creates A, B, and C
// links them together

A = new object();
A.child = B;
A.child = C;
// ...

Java Virtual Machine #1
Application instance

Java Virtual Machine #2
Application instance

A
B
C
Creating distributed garbage
Mutating that graph

Java Virtual Machine #1
Application instance

Java Virtual Machine #2
Application instance

// later, #2 changes C to E
//...
A.setChild( E );
// ...
Creating distributed garbage
Mutating that graph

Java Virtual Machine #1
Application instance

Java Virtual Machine #2
Application instance

A
B
C
E

// later, #2 changes C to E
//...
A.setChild( E );
// ...
“C” becomes distributed garbage, even though JVM #2 never accesses it.
Agenda

- Hypothesis
- Case study: virtual heaps / distributed gc
- Terminology
- **Step 1**: DGC in single Terracotta server
- **Step 2**: Adding young generations
- **Step 3**: DGC for Terracotta server arrays
- Conclusions
DGC in a single Terracotta server

> Concurrent mark / sweep algorithm
> Phases: mark, rescue1, rescue2, delete
> **Mark** objects still reachable from the set of all roots (**rootset**)
> monitor references in app heap for deltas
> rescue any objects not marked but joining the graph during mark phase (**rescueset**)
> rescue any objects not reachable but still in L1 heap
> Pause L2, rescue again, and delete garbage
Marking reachable objects
GC_Candidates exclude rootset, reachables

GC_Candidates:
A B C D E F G L M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:
A B C D E F G L M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:
B C D E F G L M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:
C D E F G L M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:
C E F G L M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates: C F G L M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L
objects paged in to L1 heap

GC_Candidates:
F G L M N O P

Sunday, 7 June 2009
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:

GLMNOP
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

GC_Candidates:
L M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachableables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:
M N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:

N O P
Marking reachable objects
GC_Candidates exclude rootset, reachables

**rootset:** A, L

**GC_Candidates:**

OP
Marking reachable objects
GC_Candidates exclude rootset, reachables

rootset: A, L

objects paged in to L1 heap

GC_Candidates:
P
Rescue phase 1
Reference monitoring informs L2 of changes
Rescue phase 1
Reference monitoring informs L2 of changes

objects paged in to L1 heap

A
B C
D E F G
L
M N O

reference deleted
Rescue phase 1
Deleted references that are still paged in are special
Rescue phase 1
Deleted references that are still paged in are special
Rescue phase 1
Because they can be rescued by other paged in objects
Rescue phase 1
Because they can be rescued by other paged in objects

objects paged in to L1 heap
A
B C
D E F G
L
M N OP

reference rescued
Rescue phase 1
The resultant garbage in this scenario

GC_Candidates:
G    P
Rescue phase 2
One quick pause and we are done

> Logically same as rescue 1
> but no changes are allowed into L2
  • “C” could not be collected even if not rescued because it cannot be GCed
> So apply any rescueset objects to GC_Candidates
> Then collect garbage
> In the example, “G”, “P” are garbage
> We’re done...
Drawbacks to simple DGC
Might require tuning

> Frequent DGC
  • pauses app too often
  • uses up L2 cycles / adds workload

> Infrequent DGC
  • more time to delete
  • increase chance that reachable objects have flushed to disk
  • disk can fill up
  • more overhead in bookkeeping garbage
Agenda

> Hypothesis
> Case study: virtual heaps / distributed gc
> Terminology
> Step 1: DGC in single Terracotta server
> **Step 2: Adding generational collection**
> Step 3: DGC for Terracotta server arrays
> Conclusions
Generational collector trying to avoid distribution

> If we partition data across L2s, each will have less data to manage
  • divide and conquer data
  • divide and conquer GC
  • ...but more later

> Generational collection
  • We can avoid marking anything that is on disk
  • can help avoid disk I/O
  • maybe help avoid the need to distribute L2s
Generational collection
Differences from simple DGC

> Similar to simple DGC
> redefine rootset to be roots currently in memory
> add oldgen, younggen spaces
  • oldgen \rightarrow objects that have fallen out of cache at some point
  • younggen \rightarrow newly created objects in L2
> rememberedset
  • backpointers from oldgen to younggen
  • this one is tricky
Young generation pictorial
oldgen, younggen, rootset
Young generation pictorial
oldgen, younggen, rootset

rootset: A

younggen in memory
oldgen on disk
The rememberedset
backpointers help skip I/O

rootset: A
rememberedset: C, M
The rememberedset
Let’s mark in younggen

rootset: A
rememberedset: C, M

GC_Candidates:
A B C  M  P
The rememberedset
Let’s mark in younggen

rootset: A
rememberedset: C, M

GC_Candidates:
B C M P
The rememberedset
Let’s mark in younggen

rootset: A
rememberedset: C, M

GC_Candidates:
C    M    P
The rememberedset
Let’s mark in younggen

rootset: A
rememberedset: C, M

GC_Candidates:
C   M   P
The rememberedset
Let’s mark in younggen

GC_Candidates:

M P
The rememberedset
Let’s mark in younggen

rootset: A
rememberedset: C,M

DISK BARRIER

GC_Candidates:

P
Generational collection summary
We need a generic “large scale” algorithm

> Benefits
• We avoided all disk I/O
• Younggen GC run in milliseconds
• Can run very frequently
• avoids sending objects to oldgen / disk

> Issues
• Doesn’t visit all objects, doesn’t find all garbage
• Full DGCs must still occur
• Distributed garbage must be created often
Agenda

> Hypothesis
> Case study: virtual heaps / distributed gc
> Terminology
> Step 1: DGC in single Terracotta server
> Step 2: Adding generational collection
> Step 3: DGC for Terracotta server stripes
> Conclusions
DGC for Server Stripes
lots of challenges to address

> Distributing the object graphs lets us handle arbitrary use cases

> We can do fast updates w/o distributed transaction overhead
  • another talk though…

> But we can’t get around distributed algorithm for GC
  • no single L2 knows the whole graph shape
  • what if nodes fail? network fails?
  • async updates means DGC is totally async
Graphical view of server stripes

Terracotta server 1  Terracotta server 2  Terracotta server 3

A  B  C  D  E  F  G  L  M  N  O  P  M  N  O

Sunday, 7 June 2009
Core algorithm

- Coordinator L2 decides when to start a DGC
- Coordinator L2 decides when it is complete
- pass marking token on to other L2s
- Share no lists, sets, etc. amongst L2s

- Make L2s respond to a global ticker event to track progress
Distributed marking

Terracotta server 1
- rootset: A
- GC_Candidates: A B D E M

Terracotta server 2
- rootset: null
- GC_Candidates: C F G N

Terracotta server 3
- rootset: L
- GC_Candidates: L O P
Distributed marking

Terracotta server 1
- rootset: A
- GC_Candidates: D E M

Terracotta server 2
- rootset: null
- GC_Candidates: C F G N

Terracotta server 3
- rootset: L
- GC_Candidates: L O P

Diagram:
- Nodes A, B, C, D, E, F, G, M, N, O
- Connections: A → B, B → D, B → E, C → F, C → G, M → N, N → O
Distributed marking

Terracotta server 1
rootset: A
GC_Candidates: E M

Terracotta server 2
rootset: null
GC_Candidates: C F G N

Terracotta server 3
rootset: L
GC_Candidates: L O P

A

B

D

E

C

F

G

M

N

O

P

Sunday, 7 June 2009
Distributed marking

Terracotta server 1
rootset: A
GC_placements: M

Terracotta server 2
rootset: null
GC_placements: C_F_G_N

Terracotta server 3
rootset: L
GC_placements: L_O_P
Distributed marking

Terracotta server 1

rootset: A
GC_Candidates: M

Terracotta server 2

rootset: null
GC_Candidates: C F G N

Terracotta server 3

rootset: L
GC_Candidates: L O P

outbound mark request

A

B

D

E

C

F

G

M

N

O

L

P

Sunday, 7 June 2009
Distributed marking

Terracotta server 1
- rootset: A
- GC_Candidates: M

Terracotta server 2
- rootset: null
- GC_Candidates: F G N

Terracotta server 3
- rootset: L
- GC_Candidates: L O P

D B C M
E F G N
L P O

outbound mark request
Distributed marking

Terracotta server 1
- rootset: A
- GC_Candidates: M

Terracotta server 2
- rootset: null
- GC_Candidates: G, N

Terracotta server 3
- rootset: L
- GC_Candidates: L, O, P

outbound mark request

Sunday, 7 June 2009
Distributed marking

Terracotta server 1
rootset: A
GC_Candidates: M

Terracotta server 2
rootset: null
GC_Candidates: N

Terracotta server 3
rootset: L
GC_Candidates: L O P

A
B
C
D
E
F
G
M
N
O

outbound mark request

Sunday, 7 June 2009
Distributed marking

**Terracotta server 1**
- rootset: A
- GC_Candidates: M

**Terracotta server 2**
- rootset: null
- GC_Candidates: N

**Terracotta server 3**
- rootset: L
- GC_Candidates: O P

Distributed marking diagram with nodes A, B, C, D, E, F, G, M, N, O, L, and P, showing outbound mark request.
Distributed marking

Terracotta server 1
rootset: A
GC_Candidates: M

Terracotta server 2
rootset: null
GC_Candidates: N

Terracotta server 3
rootset: L
GC_Candidates: O

outbound mark request

Sunday, 7 June 2009
DGC for stripes details

> Mark starts at roots
  • Requests to mark are batched, asynch for “A”, “C” and “L”

> We find the “M,N,O” graph to be garbage

> Since no single machine has all GC_Candidates we…
  • Asked Server #2 to follow from “C” for us

> We ignore failures and abandon the DGC run
  • Nothing persisted / changed till last stage
  • will get results when it comes back to life
The TICKER TOKEN
our coordinating device

> Ticker token ➔ a token that gets passed around the server ring

> Coordinator initiates

> Each node adds its status and passes token on
  • outbound ➔ farmed marking tasks to other servers
  • inbound ➔ tasks completed on behalf of others

> Phase complete when \( \Sigma \text{inbound} - \Sigma \text{outbound} = 0 \)

> Phase not complete, start a new token on next tick
DGC for stripes is totally asynchronous

> We do...
  • mark in parallel
  • rescue in parallel
  • accept graph changes in parallel

> We don’t...
  • share GC_Candidates, rootset or rememberedset
  • have any synchronous remote calls
Platform independent

> We could have used
  • messaging
  • JGroups
  • RMI
  • raw TCP or UDP protocols
  • RESTful HTTP interface
> Transport doesn’t really matter.
> and neither does language or framework, we think
Have we proven our hypothesis?

> Algorithm design outweighs platform importance
> Specific asynch frameworks may or may not help
> Not a dig on any technology but
  • the TICKER TOKEN
  • the mark/sweep
  • and more, make us go fast and those are algorithms
> Was distribution valuable? On 10GB of data:
  • 1 L2: 2.2MM objects, 1 hour
  • 8 L2s: 2.2MM objects, 55 seconds
Future optimizations

> Generational collection for stripes
> Object migration and packing, train algorithms
> Resume after crash (only restarts today)
Want to learn more?

> Come by our booth
> Ask about
  - avoiding distributed transactions
  - lock leasing and fairness
> There are all sorts of cool runtime optimizations inside the Terracotta array
> [http://www.terracotta.org](http://www.terracotta.org)
> TWITTER: @ikarzali @ssubbiah
Ari Zilka, Saravanan Subbiah
ari@terracotta.org
ssubbiah@terracotta.org

@ikarzali
@ssubbiah
@terracottatech