Decentralized Distributed Garbage Collection using Automatic Reference Counting

Tyler Leben
Team UND
lebents@gmail.com

The goal of this proposal is to: *i*) explore DCG (distributed garbage collection) techniques, *ii*) adapt and combine existing memory management algorithms to optimally work with a distributed language, *iii*) test and implement distributed garbage collection in the language SALSA Lite(Simple Actor Language System and Architecture), *iv*) evaluate against current approach.

Problem: In distributed computing, DCG is inefficient and complex due to the process of tracking local and global states of the distributed system. Inefficiency stems from performing garbage collection while executing the program. Program execution can either halt or slow down while garbage collection is being performed, so a fast and efficient algorithm needs to be implemented in DCG.

Languages such as SALSA Lite choose to omit this feature to maintain speed and efficiency. SALSA Lite compiles to Java and runs on the standard JRE (Java Runtime Environment) but cannot take advantage of Java's garbage collection because a single SALSA Lite program can run on multiple theaters (hosts) containing multiple stages (JRE runtime thread) and expand/shrink the number of theaters or stages utilized dynamically and independent of source code.

Example of Problem: Figure 1 shows the example SALSA Lite program HelloWorld.salsa. This program prints "Hello new World!" to standard output, then must be terminated by the user. The program does not finish on it's own because the StandardOutput actor can have references to other actors and also remains reachable by other threads. In other words, the StandardOutput actor is always waiting for messages and therefore remains live while maintaining reference to it's stage. In this case, Java's built-in garbage collection can not remove all references and exit.

Current Approach: SALSA 1.X.X (SALSA Lite contains no DCG) contains an local garbage collection and an optional DCG service. The algorithm for local garbage collection uses a snapshot-based, pseudo root based approach in order to get a snapshot of the meaningful global state. This works on the basis that that every actor has persistence references to root actors and an unblocked actor is a root actor. This takes a passive approach to garbage collection. The distributed garbage collection handles acyclic distributed garbage collection and local garbage collection. DCG invokes a centralized service that manages the distributed theaters by requesting local meaningful global snapshots. The centralized service identifies garbage and sends a message containing a garbage list. This process repeats itself at a user defined interval, with 20 seconds being the minimum before performance is degraded.

Problems with the above approach include the time required and having a centralized service. A program running in a distributed environment can change greatly in 20 seconds. Actors can migrate theaters for many reasons including the addition/reduction of resources. So, if one host drops out of the distributed system, it's stage and actors can migrate to another theatre but if we lose the host running the centralized service, all garbage collection stops and may or may not affect the remainder of the task. This puts an unnecessary constraint on a distributed system by using a centralized shared resource.

Approach: DCG techniques tend to be overly complex and decrease program execution speed. A combination of reference counting and Java's built-in garbage collection will be used. The addition of a reference count to the stages and actors in a SALSA Lite program will eliminate the need to maintain a complex snapshot of the distributed system. The expected results are a simple and near real-time dynamic garbage collection system that does not depend on a centralized shared service or resource. The language SALSA Lite will be used to implement the DCG algorithm and will be evaluated by comparing the speed and efficiency against the current DCG approach. Success will be based on: *i*) the elimination of the required centralized garbage collection service and *ii*) both speed and efficiency, remaining the same or improving when compared to the current approach.

About SALSA Lite: SALSA Lite 2.X.X (Simple Actor Language System and Architecture) is an implementation of the actor model designed to utilize the advantages of object-oriented programming. SALSA Lite is compiled to Java and runs in the standard JRE. Abstractions allowing development of dynamic reconfigurable programs include active objects, asynchronous message passing, universal naming, migration, and advanced coordination constructs for concurrency. A SALSA Lite program consists of one or more distributed nodes containing a theatre (host), a stage (JRE runtime thread) and universal actors that can be passed around the nodes. Figure 1 shows an example SALSA Lite program and figure 2 shows the resultant Java file.

Figure 1:

```
//HelloWorld.salsa
//Salsa introduces synchronization of messages using @
//@ guarantees the method before the @ will be processed before
//the message afterward
import salsa_lite.io.StandardOutput;

behavior HelloWorld {
   HelloWorld(String[] arguments){
      StandardOutput standardOutput = new StandardOutput(); //Actor
  //messages
      standardOutput<-print("Hello") @
      standardOutput<-println(" World!");

      standardOutput<-print(" new " );
      //prints Hello new World! because the message 'print("...")@'
      // (1)is guaranteed to be processed first and
      // (2) the StandardOutput actor expects another 'print("...")'
         message
      // before processing any other messages
   }
   }
```

Figure 2:

```
 //HelloWorld.java
/****** SALSA LANGUAGE IMPORTS ******/
import salsa_lite.common.DeepCopy;
import salsa_lite.runtime.LocalActorRegistry;
import salsa_lite.runtime.Hashing;
import salsa_lite.runtime.Acknowledgement;
import salsa_lite.runtime.SynchronousMailboxStage;
import salsa_lite.runtime.Actor;
import salsa_lite.runtime.Message;
import salsa_lite.runtime.RemoteActor;
import salsa_lite.runtime.MobileActor;
import salsa_lite.runtime.StageService;
import salsa_lite.runtime.TransportService;
import salsa_lite.runtime.language.Director;
import salsa_lite.runtime.language.JoinDirector;
import salsa_lite.runtime.language.MessageDirector;
import salsa_lite.runtime.language.ContinuationDirector;
import salsa_lite.runtime.language.TokenDirector;

import salsa_lite.runtime.language.exceptions.RemoteMessageException;
import salsa_lite.runtime.language.exceptions.TokenPassException;
import
   salsa_lite.runtime.language.exceptions.MessageHandlerNotFoundException;
import
   salsa_lite.runtime.language.exceptions.ConstructorNotFoundException;

/****** END SALSA LANGUAGE IMPORTS ******/

import salsa_lite.runtime.io.StandardOutput;

public class HelloWorld extends salsa_lite.runtime.Actor implements
   java.io.Serializable {


   public Object writeReplace() throws java.io.ObjectStreamException {
      int hashCode = Hashing.getHashCodeFor(this.hashCode(),
         TransportService.getHost(), TransportService.getPort());
      synchronized (LocalActorRegistry.getLock(hashCode)) {
         LocalActorRegistry.addEntry(hashCode, this);
      }
      return new SerializedHelloWorld( this.hashCode(),
         TransportService.getHost(), TransportService.getPort() );
   }
```

```java
public static class RemoteReference extends HelloWorld {
   private int hashCode;
   private String host;
   private int port;
   RemoteReference(int hashCode, String host, int port) {
      this.hashCode = hashCode; this.host = host; this.port = port;
      }

   public Object invokeMessage(int messageId, Object[] arguments)
      throws RemoteMessageException, TokenPassException,
      MessageHandlerNotFoundException {
      TransportService.sendMessageToRemote(host, port,
         this.getStage().message);
      throw new RemoteMessageException();
   }

   public void invokeConstructor(int messageId, Object[] arguments)
      throws RemoteMessageException, ConstructorNotFoundException {
      TransportService.sendMessageToRemote(host, port,
         this.getStage().message);
      throw new RemoteMessageException();
   }

   public Object writeReplace() throws
      java.io.ObjectStreamException {
      return new SerializedHelloWorld( hashCode, host, port);
   }
}

public static class SerializedHelloWorld implements
   java.io.Serializable {
   int hashCode;
   String host;
   int port;

   SerializedHelloWorld(int hashCode, String host, int port) {
      this.hashCode = hashCode; this.host = host; this.port = port;
      }

   public Object readResolve() throws java.io.ObjectStreamException
      {
      int hashCode = Hashing.getHashCodeFor(this.hashCode,
         this.host, this.port);
      synchronized (LocalActorRegistry.getLock(hashCode)) {
         HelloWorld actor =
            (HelloWorld)LocalActorRegistry.getEntry(hashCode);
            if (actor == null) {
                RemoteReference remoteReference = new
                    RemoteReference(this.hashCode, this.host, this.port);
                LocalActorRegistry.addEntry(hashCode, remoteReference);
                return remoteReference;
            } else {
                return actor;
            }
        }
    }
}

public String getMessageInformation(int messageId) {
  switch (messageId) {
    case 0: return "java.lang.String
        [salsa_lite.runtime.Actor].toString()";
    case 1: return "int [salsa_lite.runtime.Actor].hashCode()";
    case 2: return "int [salsa_lite.runtime.Actor].getStageId()";
  }
  return "No message with specified id.";
}

public HelloWorld() { super(); }
public HelloWorld(int stage_id) { super(stage_id); }



public Object invokeMessage(int messageId, Object[] arguments)
    throws RemoteMessageException, TokenPassException,
    MessageHandlerNotFoundException {
    switch(messageId) {
        case 0: return toString();
        case 1: return hashCode();
        case 2: return getStageId();
        default: throw new MessageHandlerNotFoundException(messageId,
            arguments);
    }
}

public void invokeConstructor(int messageId, Object[] arguments)
    throws RemoteMessageException, ConstructorNotFoundException {
    switch(messageId) {
        case 0: construct( (String[])arguments[0] ); return;
        default: throw new ConstructorNotFoundException(messageId,
            arguments);
    }
```

```
}

public void construct() {}

public void construct(String[] arguments) {
   StandardOutput standardOutput = StandardOutput.construct(0,
      null);
   ContinuationDirector continuation_token =
      StageService.sendContinuationMessage(standardOutput, 3
      /*print*/, new Object[]{"Hello"});
   StageService.sendMessage(standardOutput, 13 /*println*/, new
      Object[]{" World!"}, continuation_token);
   StageService.sendMessage(standardOutput, 3 /*print*/, new
      Object[]{" new "});
}




public static void main(String[] arguments) {
   HelloWorld.construct(0, new Object[]{arguments});
}

public static HelloWorld construct(int constructor_id, Object[]
   arguments) {
   HelloWorld actor = new HelloWorld();
   StageService.sendMessage(new Message(Message.CONSTRUCT_MESSAGE,
      actor, constructor_id, arguments));
   return actor;
}

public static HelloWorld construct(int constructor_id, Object[]
   arguments, int target_stage_id) {
   HelloWorld actor = new HelloWorld(target_stage_id);
   actor.getStage().putMessageInMailbox(new
      Message(Message.CONSTRUCT_MESSAGE, actor, constructor_id,
      arguments));
   return actor;
}

public static TokenDirector construct(int constructor_id, Object[]
   arguments, int[] token_positions) {
   HelloWorld actor = new HelloWorld();
   TokenDirector output_continuation = TokenDirector.construct(0
      /*construct()*/, null);
   Message input_message = new
```

```
            Message(Message.CONSTRUCT_CONTINUATION_MESSAGE, actor,
            constructor_id, arguments, output_continuation);
         MessageDirector md = MessageDirector.construct(3, new
            Object[]{input_message, arguments, token_positions});
         return output_continuation;
      }

      public static TokenDirector construct(int constructor_id, Object[]
         arguments, int[] token_positions, int target_stage_id) {
         HelloWorld actor = new HelloWorld(target_stage_id);
         TokenDirector output_continuation = TokenDirector.construct(0
            /*construct()*/, null, target_stage_id);
         Message input_message = new
            Message(Message.CONSTRUCT_CONTINUATION_MESSAGE, actor,
            constructor_id, arguments, output_continuation);
         MessageDirector md = MessageDirector.construct(3, new
            Object[]{input_message, arguments, token_positions},
            target_stage_id);
         return output_continuation;
      }

   }
   }
```