Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Alen Vrečko

Odročno nalaganje razredov za knjižnico Akka

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: izr. prof. dr. Marko Bajec

Ljubljana 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00201/2012
Datum: 05.03.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ALEN VREČKO**

Naslov: **ODROČNO NALAGANJE RAZREDOV ZA KNJIŽNICO AKKA**
REMOTE CLASS LOADING FOR AKKA LIBRARY

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

AKKA je orodje za izdelavo kompleksnih aplikacij, ki potrebujejo visoko raven sočasnosti, distribucije in razširljivosti (ang. Scalability). Kadar želimo integrirati več s knjižnico Akka razvitih rešitev, je moteče to, da moramo javine programske razrede ročno kopirati med računalniki, na katerih posamezne rešitve delujejo. V okviru diplomske naloge najprej bralcu predstavite knjižnico Akka, nato pa preučite, kako bi lahko implementirali mehanizem za avtomatsko nalaganje razredov. Izberite eno od možnosti ter jo implementirajte.

Mentor:

prof. dr. Marko Bajec

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Alen Vrečko, z vpisno številko **63050133**, sem avtor diplomskega dela z naslovom:

Odročno nalaganje razredov za knjižnico Akka

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Marka Bajca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 13. junija 2012 Podpis avtorja:

Kazalo

Seznam uporabljenih kratic in simbolov

- JVM - Javin navidezni stroj (ang. Java Virtual Machine)
- CAS - primerjaj in zamenjaj (ang. Compare and Swap)

Povzetek

Cilj tega dela je implementacija odročnega nalaganja razredov za knjižnico Akka. Knjižnica Akka omogoča programiranje sočasnosti v modelu igralcev na Javinem navideznem stroju (ang. Java Virtual Machine). Omogoča enostavno skalabilnost. Podan problem lahko rešimo hitreje tako, da povečamo število igralcev in strojno moč t.i. *scaling up*. Pohitrimo pa ga lahko tako, da v reševanje problema vključimo več fizično ločenih računalnikov t.i. *scaling out*.

Pri slednjem pride naše diplomsko delo do izraza. Definicija igralcev ter sporočil je namreč zapisana v vmesni kodi za JVM (ang. bytecode). Če želimo naš sistem igralcev povezati z odročnim sistemom moramo sami poskbreti, da so definicije sporočil ter igralcev na voljo na obeh sistemih.

Naše delo poskrbi, da se definicije samodejno prenesejo. Problem deluje zelo preprosto, vendar je veliko malenkosti na katere moramo biti pozorni. Odročno nalaganje lahko opravimo na več načinov. Odločili smo se, da odročno nalaganje kode izvaja nalagalec razredov (ang. Class Loader). Pri tem blokira izvajanje, dokler ne naloži definicije razreda.

Ključne besede:

distribuirani sistem, model igralcev, nalaganje kode, vmesna koda, akka

Abstract

The goal of the thesis is to implement Remote Class Loading for the Akka library. Akka library provides actor model of concurrency for the Java Virtual Machine. It provides easy scalability. We can scale the problem by *scaling up* i.e. using more actors and providing more hardware resources for the JVM. Alternatively we can also *scale out* by providing more JVMs (on separate machines).

By the latter our work helps. Actors and messages are defined in bytecode. When we want to connect our actor system to a remote system we need to manually guarantee that both sistems have the same bytecode available.

Our work makes class definitions available without manual intervention. The problem looks simple but there are numerous details that need to be worked out. There are also numerous ways of implementing this functionality. We chose to trigger the remote class loading functionality in the Class Loader. It blocks the executing thread until it receives the class definitions.

Keywords:

distributed sistem, actor model, class loading, bytecode, akka

Poglavje 1

Uvod

Ideja za delo je prišla iz istoimenskega projeka pri Googlovem poletju kode. V času izvajanja poletja kode je bila knjižnica Akka v verziji 1.x. V omenjeni verziji je bilo nemogoče dodati omenjeno funkcionalnost, brez drastičnega posega v obstoječo kodo. Zato tudi na koncu funkcionalnost nikoli ni postala del Akke.

V okviru diplomske naloge smo implementirali funkcionalnost na podlagi verzije 2.x, ki je drastično bolj razširljiva in smo lahko implementirali funkcionalnost brez posega v obstoječo izvorno kodo.

Naša implementacija čaka na vključitev v knjižnico Akko, ko v okviru odprtokodnega procesa odpravimo še zadnje malenkosti.

Preden lahko opišemo naše delo, se nam zdi pomembno, da se predstavi na kratko model igralcev v poglavju 3. Preden pa lahko opišemo model igralcev se nam zdi smiselno, da na kratko opišemo obstoječe načine za sočasnost. To smo naredili v poglavju 2.

Ko razumemo model igralcev, lahko razumemo, zakaj potrebujemo odročno nalaganje razredov. Sam problem je natančno opisan v poglavju 4.

Sama implementacija je natančno opisana v poglavju 6. Za testiranje smo razvili posebno knjižnico imenovano path-hole, ki je opisana v poglavju 5, ki govori o testiranju.

Sklepne ugotovitve smo zapisali v zadnjem poglavju.

Poglavje 2

Sočasnost na JVMju

Sočasnost ne pomeni samo, da se izvajata vsaj dve operaciji istočasno, ampak da tudi tekmujeta za dostop do resursov. Tekmovani resurs je lahko baza, datoteka ali največkrat kar lokacija v spominu.

Bistvo sočasnega izvajanja kode sta dve stvari, in sicer medsebojno izključevanje ter vidnost sprememb. Medsebojno izključevanje pomeni nadzorovanje tekmovanih sprememb istega resoursa. Vidnost sprememb je nadzorovanje, kako so dane spremembe vidne ostalim operacijam.

Poglejmo si konkretno, kako je s sočasnostjo na javinem navideznem stroju.

2.1 Nitenje

Specifikacija jezika Jave [1], v poglavju 17, definira niti in ključavnice kot osnovo za sočasno programiranje. Za občutek sprogramirajmo enostaven primer borze.

Imamo borzo, ki trguje samo delnice fiktivne družbe *Foo*. Sočasno hočemo izvajati operacijo kupovanja nakjučne vrednosti delnice ter operacijo prodaje delnice.

Potrebujemo razred `FooStock`, ki bo predstavljal stanje delnic družbe *Foo*. Ter dve niti, eno, ki kupuje in drugo, ki prodaja. Program lahko zaženemo z

razredom `StockExchange`.

```
1 public class FooStock {
2
3     private int nrOfStocks;
4
5     public FooStock(int nrOfStocks) {
6         this.nrOfStocks = nrOfStocks;
7     }
8
9     public synchronized boolean buy(int count) {
10        System.out.println("buy stock = " + nrOfStocks);
11        if (nrOfStocks >= count) {
12            nrOfStocks -= count;
13            return true;
14        } else return false;
15    }
16
17    public synchronized void sell(int count) {
18        System.out.println("sell stock = " + nrOfStocks);
19        nrOfStocks += count;
20    }
21
22 }
```

```
1 public class Seller implements Runnable {
2
3     private FooStock stock;
4
5     public Seller(FooStock stock) {
6         this.stock = stock;
7     }
8
9     public void run() {
10        try {
11            while (!Thread.interrupted()) {
12                stock.sell((int) (Math.random() * 100));
```

```
13                 Thread.sleep(10);
14             }
15         } catch (InterruptedException e) {
16             // end execution
17         }
18     }
19 }
```

```
1 public class Buyer implements Runnable {
2
3     private FooStock stock;
4
5     public Buyer(FooStock stock) {
6         this.stock = stock;
7     }
8
9     public void run() {
10        try {
11            while (!Thread.interrupted()) {
12                stock.buy((int) (Math.random() * 100));
13                Thread.sleep(10);
14            }
15        } catch (InterruptedException e) {
16            // end execution
17        }
18    }
19 }
```

```
1 public class StockExchange {
2
3     public static void main(String[] args) {
4         FooStock fooStock = new FooStock(10000);
5         new Thread(new Buyer(fooStock)).start();
6         new Thread(new Seller(fooStock)).start();
7     }
8
9 }
```

Kot vidimo, obe niti sočasno tekmujeta za polje `nrOfStocks` na `FooStock`. To predstavlja t.i. kritično sekcijo. Kritično sekcijo smo zaščitili s pomočjo uporabe *synchronized* metode.

Kot vidimo na tem enostavnem primeru, programiranje na tak način zahteva, da razmislimo o kritičnih sekcijah in ustrezno uredimo medsebojno izključevanje dostopa niti.

Vendar je tu še problem vidnosti. Poglejmo si primer iz knjige Effective Java [2].

```
1 import java.util.concurrent.TimeUnit;
2
3 public class StopThread {
4     private static boolean stopRequested;
5
6     public static void main(String[] args) throws
          InterruptedException {
7         Thread backgroundThread = new Thread(new Runnable() {
8             public void run() {
9                 int i = 0;
10                while (!stopRequested)
11                    i++;
12            }
13        });
14        backgroundThread.start();
15        TimeUnit.SECONDS.sleep(1);
16        stopRequested = true;
17    }
18 }
```

Pričakovali bi, da se zgornji program izvaja 1 sekundo, nato pa se ustavi. Vendar ni tako. V zgornjem primeru vidimo, da samo nit main metode spremeni vrednost spremenljivke `stopRequested`. Nit `backgroundThread` pa samo bere to spremenjlivko. Naivno bi pričakovali, da ne pride do konfliktov in bo druga nit že videla spremembe, ki jih je naredila prva. Ko zaženemo zgornji program se ta nikoli ne ustavi. Imamo problem vidnosti, in sicer spremembe,

ki jih je naredila nit main metode niso vidne niti `backgroundThread`.

To je zato, ker je JVM naredil optimizacijo kode, in sicer ukaze je iz

```
while (!done)
i++;
```

prepisal v

```
if (!done)
while(true)
i++;
```

Kot vidimo, ne pride do ponovnega branja spremenljivke. Problem rešimo tako, da polje `stopRequested` definiramo kot `volatile`. Volatile povzroči spominsko bariero, ki garantira vidnost spremembe spremenljivke. To pomeni, da se polje vedno sinhronizira z glavnim spominom.

Kot vidimo, pri delu z nitmi ni dovolj, da razmišljamo o medsebojnemu izključevanju, ampak moramo razmišljati tudi o vidnosti sprememb. Naj omenimo še, da uporaba zaklepanja povzroči tudi spominsko bariero. Tako, da stvari znotraj sinhronizacijskih konstruktov ne potrbujemo definirati kot volatile.

Dani primer bi lahko popravili tudi tako, da do polja dostopamo samo znotraj synchronized bloka ali uporabimo razred `java.util.concurrent.ReentrantLock`. Vendar to poleg spominske bariere tudi blokira nit. Sama uporaba volatile pa ne blokira niti.

2.2 CAS

Poleg eksplicitnega zaklepanja kritične sekcije imamo na voljo t.i. primerjaj in zamenjaj (ang. Compare-And-Swap). Ideja je, da atomično spremenimo vrednost spremeljivke pod pogojem, da se ujema vrednosti, ki smo jo prebrali. Poglejmo si ponovno naš primer borze.

Namesto uporabe synchronized metode za buy in sell, bi lahko stvar napisali takole:

```
1 import java.util.concurrent.atomic.AtomicInteger;
2
3 public class FooStockCas {
4
5     private AtomicInteger nrOfStocks;
6
7     public FooStockCas(AtomicInteger nrOfStocks) {
8         this.nrOfStocks = nrOfStocks;
9     }
10
11    public int get(){ return nrOfStocks.get();}
12
13    public boolean buy(int amount, int observedCount){
14        int newNrOfStocks = observedCount -  amount;
15        if(newNrOfStocks >= 0){
16            return nrOfStocks.compareAndSet(observedCount,
                  newNrOfStocks);
17        }
18        else return false;
19    }
20
21    public boolean sell(int amount, int observedCount){
22        return nrOfStocks.compareAndSet(observedCount,
              observedCount + amount);
23    }
24
25 }
```

```
1 public class SellerCas implements Runnable {
2
3     private FooStockCas stock;
4
5     public SellerCas(FooStockCas stock) {
6         this.stock = stock;
7     }
8
9     public void run() {
```

```
10          try {
11              while (Thread.interrupted()) {
12                  int observed = stock.get();
13                  int sellAmount = (int) (Math.random() * 100);
14                  stock.sell(sellAmount, observed);
15                  Thread.sleep(10);
16              }
17          } catch (InterruptedException e) {
18              // stop
19          }
20      }
21
22  }
```

```
1  public class BuyerCas implements Runnable {
2
3      private FooStockCas stock;
4
5      public BuyerCas(FooStockCas stock) {
6          this.stock = stock;
7      }
8
9      public void run() {
10         try {
11             while (Thread.interrupted()) {
12                 int observed = stock.get();
13                 int buyAmount = (int) (Math.random() * 100);
14                 stock.buy(buyAmount, observed);
15                 Thread.sleep(10);
16             }
17         } catch (InterruptedException e) {
18             // stop
19         }
20     }
21
22 }
```

```
1 import java.util.concurrent.atomic.AtomicInteger;
2
3 public class StockExchangeCas {
4
5     public static void main(String[] args) {
6         FooStockCas fooStock = new FooStockCas(new AtomicInteger
              (10000));
7         new Thread(new BuyerCas(fooStock)).start();
8         new Thread(new SellerCas(fooStock)).start();
9     }
10
11 }
```

Kot vidimo, je že za najbolj enostavne primere programiranje z nitmi in eksplicitnim zaklepanjem ali CAS zahtevno. Predvsem, ko hočemo sprogramirati kaj bolj zahtevnega, postane koda zelo kompleksna in je izjemno težko dokazati, da deluje pravilno.

Poglejmo si alternativo, ki jo nudi knjižnica Akka s t.i. modelom igralcev.

Poglavje 3

Akka in model igralcev

Model igralcev je prvič omenjen leta 1973 v članku Carla Hewitta [3]. Populariziral ga je programski jezik Erlang.

Igralci so objekti, ki enkapsulirajo stanje in obnašanje. Med seboj komunicirajo samo tako, da si izmenjujejo sporočila, ki jih dajo v naslovnikov poštni nabiralnik.

En igralec ni noben igralec. Potrebujemo več igralcev. Igralce ustvarimo in nadzorujemo v okviru sistema igralcev.

Model igralcev definira tri aksiome. Ko igralec prejme sporočilo, sme narediti sledeče:

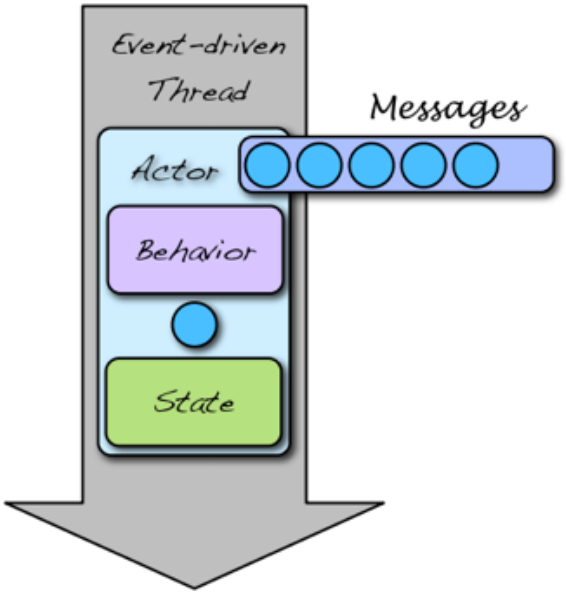
1. ustvari nove igralce,

2. poslati sporočilo drugim igralcem, (lahko tudi sebi),

3. spremeni stanje oziroma obnašanje, ki bo vidno naslednjemu sporočilu.

Pomembno je, da razumemo, da igralec lahko obdela samo eno sporočilo hkrati. Implementacije smejo to optimizirati, vendar se igralec nikakor ne sme obnašati drugače, kot bi se sicer.

Pomembno je tudi, da razumemo, da do igralca ne smemo dostopati direktno ampak samo preko naslova. Podobno kot do Googlovega fizičnega strežnika



Slika 3.1: Koncept igralca predstavljen grafično

dostopamo preko `http://www.google.com/`, dostopamo do igralcev preko njihovih naslovov. To omogoča, da za istim naslovom stoji druga instanca igralca. Točne detajle pustimo implementacijam.

3.1 Akka

Akka je odprtokodna knjižnica napisana v Scali in Javi, in nam omogoča programiranje v modelu igralcev. Akko lahko uporabljamo tako iz Scale kot iz Jave. Načeloma bomo predstavljali kodo napisano v Scali.

Poglejmo si problem borze, narejen s pomočjo knjižnice Akke, napisan v programskem jeziku Scala.

Najprej definiramo sporočila, ki si jih bodo igralci pošiljali.

```
1 case class Buy(amount:Int)
2 case class Sell(amount:Int)
3 case object Success
4 case object Fail
```

```
5 case object StartTrading
```

Kot vidimo nam programski jezik Scala omogoča, da prihranimo kar nekaj vrstic z uporabo case in object besed. Case razred je poseben razred, ki implicitno definira lastnost, ki je zapisana med oklepaji. Ne potrebujemo definirati polja ter t.i. getter in setter metode, kot je to praksa v Javi ampak to Scala prevajalnik naredi za nas. Object razred je pa implicitno definiran kot t.i. singleton kar pomeni, da obstaja samo ena instanca tega razreda. To je Scalin odgovor na statične metode in polja v Javi.

Poglejmo si še celoten primer borze narejen v programskem jeziku Scala z uporabo knjižnice Akka.

```
1 import akka.event.Logging
2 import akka.actor.{ActorRef, Props, ActorSystem, Actor}
3
4 class FooStockActor extends Actor {
5
6   val log = Logging(context.system, this)
7
8   // igralec ima stanje v tem primeru stevilo delnic
9   // spremenljivke ne rabimo definirati kot volatile
10  // za pravilnost skrbi akka
11  var nrOfStock = 10000
12
13  // igralec ima obnasanje, ki je realizirano z
14  // metodo receive (prejmi) tukaj definiramo, kaj se zgodi
15  // ko prejmemo sporocilo
16  def receive = {
17    // ko prejmemo sporocilo Buy zmanjsamo st. delnic, ce je to
        mogoce
18    // sama sintaksa je t.i. pattern-matching funkcijskega
        programiranja,
19    // Scala sprida podpira konstrukte funkcijskega programiranja
20    case Buy(amount) => {
21      if(nrOfStock >= amount){
22        // opazimo, da upostevamo Aksiom 3 - smemo spremeniti
```

```
            stanje
23          // ki bo vidno naslednjemu sporocilu
24          nrOfStock-=amount
25          sender ! Success
26        }else {
27          sender ! Fail
28        }
29      }
30
31      // obdelamo tudi sporocilo Sell
32      case Sell(amount) => {
33        nrOfStock+=amount;
34        // opazimo, da upostevamo Aksiom 2 - Smemo poslati sporocilo
35        // in sicer posljemo sporocilo nasemu posiljatelju
36        sender ! Success
37      }
38      // v primeru, da dobimo sporocilo katerega tipa ne poznamo
            samo zapisemo v log
39      case _ => log.info("received unknown message")
40    }
41 }
42
43 // kot smo omenili object razred je poseben razred, ki je t.i.
       singleton
44 // imamo metodo main, ki je ekvivalentna javini public static void
       main
45 object Start {
46   def main(args: Array[String]) {
47     // ustvarimo nov sistem igralcev z imenom FooStockSystem
48     val system = ActorSystem("FooStockSystem")
49     // ustvarimo novega igralca tipa FooStockActor
50     // nazaj pa dobimo instanco tipa ActorRef, ki
51     // predstavlja samo nek naslov na katerm se nahaja nas igralec
52     // oz. lahko bi se nahajal tudi skupek igralcev
53     // to je kljucnega pomena, da ne dobimo direktne reference
54     // ampak dobimo referenco, ki jo lahko zlorabljamo kakor
```

```
        hocemo,
55      // posljemo cez mrezo, itd.
56      val fooStockActor:ActorRef = system.actorOf(Props[
            FooStockActor])
57      // ustvarimo igralca tipa BuyActor, ki ima referenco na
            FooStockActor
58      val buyActor:ActorRef = system.actorOf(Props{
59        new BuyActor(fooStockActor)
60        })
61      // ustvarimo igralca tipa SellActor
62      val sellActor:ActorRef = system.actorOf(Props{
63        new SellActor(fooStockActor)
64        })
65      // posljemo obema igralca sporocilo StartTrading
66      buyActor ! StartTrading
67      sellActor ! StartTrading
68    }
69  }
70
71  // definicija igralca podobno kot FooStockActor
72  class BuyActor(fooStockActor:ActorRef) extends Actor {
73
74    // nimamo stanja imamo samo obnasanje
75    def receive = {
76      // ko prejmemo sporocila tipa StartTrading ali Success ali
            Fail
77      // potem posljemo novo sporocilo Buy igralcu fooStockActor
78      case StartTrading | Success | Fail => {
79        fooStockActor ! Buy((Math.random * 100).asInstanceOf[Int])
80      }
81      case _ => // drop unknown message
82    }
83
84  }
85
86  // po podobnem kopitu kot BuyActor
```

```
87 class SellActor(fooStockActor:ActorRef) extends Actor {
88
89   def receive = {
90     case StartTrading | Success | Fail => {
91       fooStockActor ! Buy((Math.random * 100).asInstanceOf[Int])
92     }
93     case _ => // drop unknown message
94   }
95
96 }
```

3.2 Omrežni sistem igralcev

Sedaj poznamo koncept modela igralcev in na grobo knjižnico Akka. Problem nalaganja razredov ni daleč proč.

Spoznali smo se z razredoma `ActorSystem` in `ActorRef`. Znotraj ActorSystem instance živijo igralci. Igralce instanciramo z `actorOf` metodo. Manjka nam pa še en kos sestavljanke, in to je, kako dobimo referenco že obstoječega igralca?

S klicem metode `actorFor` znotraj igralca ali preko ActorSystema dobimo referenco na obstoječega igralca. Primeri kode za občutek:

```
1 // naredimo sistem igralcev ter naredimo novega igralca v sistemu
2 val system = ActorSystem("NameOfSystem")
3 actorOf(Props[EchoActor], "NameOfActor")
4
5 // S klicem actorFor smo dobili lokalno referenco na igralca.
6 val local:ActorRef = actorSystem.actorFor("akka://NameOfSystem/
    user/NameOfActor")
```

Igralci so lokacijsko povsem neodvisni, zato lahko referenco na igralca serializiramo in pošljemo čez žico.

Primer:

```
1 // naredimo lokalnega igralca
```

```
2 val localActor = system.actorOf(Props[FooActor])
3
4 // dobimo referenco na odrocnega igralca
5 val remoteActor:ActorRef = actorSystem.actorFor("akka://Foo@
    192.168.1.12:5678/user/Bar")
6
7 // odrocnemu igralcu posljemo sporocilo - referenco na nasega
    igralca
8 remoteActor ! localActor
```

Sedaj se bralec najverjetneje sprašuje, kako naredimo lokalni sistem igralcev dostopen preko omrežja. Kako iz lokalnega sistema ta postane omrežni?

To naredimo s pomočjo konfiguracije. V našem projektu definiramo datoteko z imenom `application.conf`. Z vsebino:

```
akka {
    actor{
        provider="akka.remote.RemoteActorRefProvider"
    }
    remote{
        transport="akka.remote.netty.NettyRemoteTransport"
            netty{
                hostname="127.0.0.1"
                port=2552
            }
    }
}
```

Ko naredimo v naši kodi `val actorSystem = ActorSystem("Foo")`, bo le-ta vzel konfiguracijo iz `application.conf` datoteke in bo naš sistem dostopen preko omrežja na naslovu `127.0.0.1:2552`. Lahko seveda naredimo tudi usmeritve na podlagi omrežne infrastrukture podobno, kot da bi imeli aplikacijski strežnik.

Poleg tega, da dobimo referenco na odročnega igralca, lahko instanciramo igralce na odročnih sistemih preko konfiguracije, in sicer, če dopolnimo `application.conf`, da zgleda takole:

```
akka {
    actor {
        deployment {
            /foo {
                remote = "akka://FooSystem@127.0.0.1:5678"
            }
     }
}
```

Ko naredimo klic `system.actorOf(Props[FooActor],"foo")` bo Akka ustvarila igralca, ne na našem lokalnem sistemu, ampak na naslovu `akka://FooSystem@127.0.0.1:5678`. To je zaradi ujemanja imena igralca s tem v konfiguraciji.

Če bi na primer namesto tega klicali `system.actorOf(Props[FooActor],"foo2")` bi se ta igralec ustvaril lokalno, ker v konfiguraciji ni določeno, da naj se ustvari odročno.

Poglavje 4

Problem odročnega nalaganja razredov

Sedaj poznamo tako lokalni sistem, kot tudi odročni sistem igralcev. Do sedaj smo vedno predvidevali, da so vsi razredi na voljo tako na lokalnem, kot tudi na odročnem sistemu.

Da to drži, moramo ročno poskrbeti, da javina razredna pot (ang. Class Path) vsebuje vse potrebne definicije zapakirane v JAR datotekah.

V nasprotnem primeru se nam zgodi `ClassNotFoundException` na odročnem sistemu. Kar ni lepo. Naše delo poskrbi za transparentno nalaganje kode.

Želimo omogočiti naslednje:

1. pošiljanje sporočil, katere definicije niso na voljo na odročnem sistemu,
2. odročno instanciranje igralcev, keterih definicije niso na voljo na odročnem sistemu,
3. vrstni red sporočil se nikakor ne sme spremeniti,
4. interna struktura Akka knjižnice se "načeloma" ne sme spreminjati.

Poglavje 5

Testiranje

V duhu testno usmerjenga razvoja (ang. Test Driven Development) se bomo najprej lotili testov.

Da pa lahko uspešno pišemo teste, potrebujemo ustrezno infrastrukturo. Zato bomo najprej opisali potrebno posebno infrastrukturo. Pri tem smo se odličili, da ne bomo opisali splošno razširjene in splošno poznane knjižnice JUnit in Skalaških izboljšav v obliki knjižnice Scalatest.

Po opisu infrastrukture sledi primer testa.

5.1 Orodje SBT

Gre za t.i. build tool, ki nam pomaga z prevajanjem in izvajanjem Scala in Java kode. Ime je okrajšava za Simple Build Tool. Na voljo je na naslovu `http://www.scala-sbt.org/`. To orodje po našem mnenju ni pravilno imenovano. Pravo ime bi moralo biti Scala Build Tool, saj je v celoti napisano v programskem jeziku Scala, in kljub temu, da se trudi biti preprosto, to ni. Najbolj smo pogrešali prvo razredno integracijo z ravojnim okoljem IntelliJ IDEA.

Knjižnica Akka v celoti uporablja SBT kot glavni in edini build tool. Zato smo ga primorani uporabljat.

5.2 Multi-JVM Testing

Glavno testiranje se izvaja s pomočjo build toola SBT ter vtičnika Multi-JVM-Test. Vtičnik je opisan na strani dokumentacije Akke [4] .

V osnovi gre za to, da se držimo konvencije imena <ime testa>MultiJvm<ime instance JVM>.

Primer:

- `FooMultiJvmNode1.scala`
- `FooMultiJvmNode2.scala`
- `FooMultiJvmNode3.scala`

Ko zaženemo v sbt konzoli `multi:jvm:test-only Foo`, nam bo Multi-JVM-Test zagnal 3 JVM-je in main metodo oziroma testno kodo vsake istoležeče datoteke.

Kar je tu markantnega je, da vsi JVM-ji uporabljajo isti Class Path. Razred FooMultiJvmNode3 je na voljo pri izvajanju FooMultiJvmNode1 ipd. Slednje ni v redu za testiranje odročnega nalaganja razredov. Zato nam manjka še en kos sestavljanke. In to je način kako skriti določene razrede na določeni javinem navideznem stroju.

Ravno za ta del smo napisali posebno orodje imenovano path-hole. Opisnano je v sledečem podpoglavju.

5.3 Path-hole Java Agent

Gre za t.i. Java agenta, ki prisili `ClassLoader`, da vrže `ClassNotFoundException` tudi, če je razred na voljo na classpathu in bi se načeloma naložil brez problema. Mehanizem Java agenta je definiran v JSR 87 [5].

Izvorno kodo smo licencirali pod Apache License, Version 2.0 licenco. Koda je pa na voljo na naslovu `https://github.com/avrecko/path-hole`.

Uporaba je sledeča:

```
System.setProperty("path_hole.filter", "*Node2*,*Node3*")
System.setProperty("path_hole.unfiltered.cls", "akka.remote.netty.
    rcl.RemoteClassLoader")
```

S tem smo naredili razrede, katerih ime vsebuje Node2 ali pa Node3 nenaložljive na danem JVM-ju. Dovolimo pa našemu odročnemu nalagalcu razredov, da naloži te razrede.

Implementacija uporablja prej omenjen mehanizem Java agenta. Namesti se tako:

```
1 public static void premain(String agentArguments, Instrumentation
    instrumentation) {
2    // check if we can redefine the java.lang.ClassLoader
3    checkArgument(instrumentation.isRedefineClassesSupported(), "
        Path-hole agent cannot redefine the ClassLoader!");
4    // lets prepend the loadClass method with our filter
5    byte[] enhancedBytes = prependToClassLoader();
6
7    // now we can redefine it
8    try {
9       instrumentation.redefineClasses(new ClassDefinition(
            ClassLoader.class, enhancedBytes));
10   } catch (Throwable t) {
11      throw new RuntimeException(t);
12   }
13 }
```

Se pravi mi povozimo `loadClass` metodo na krovnem `ClassLoader` razredu, da vrže exception, če se razred, ki ga hočemo naložiti, ujema z definiranim filtrom.

Tu je del kode, ki ga dodamo na začetek že obstoječi load metodi na razredu `java.lang.ClassLoader`.

```
1 // we will get this method's bytecode and prepend it to the
    bytecode of java.lang.ClassLoader#loadClass(String,Bool)
```

```
2 public void bytecodeToPrepend(String name, boolean resolve) throws
      ClassNotFoundException {
3     // if we are a unfiltered classloader let us trough
4     // note: the only allowed return is at the end of this method
          due to the fact this method is prepended to j.l.CL
5     String unfilteredClassLoaders = System.getProperty(PathHole.
          UNFILTERED_CLASSLOADER_FQNS, "").trim().replace("\\s","");
6
7     boolean isUnfilteredClassLoader = false;
8
9     if (!unfilteredClassLoaders.isEmpty()) {
10        String thisFqn = this.getClass().getName();
11        for (String unfilteredClFqn : unfilteredClassLoaders.split
              (",")) {
12            if (thisFqn.equals(unfilteredClFqn.trim())) {
13                isUnfilteredClassLoader = true;
14                break;
15            }
16        }
17    }
18
19    if (!isUnfilteredClassLoader) {
20        // might not seem optimal to read properties each time but
              this allows for riches runtime behavior
21        String filter = System.getProperty(PathHole.
              FILTER_PROPERTY_NAME, "").trim();
22        if (!filter.isEmpty() && !name.startsWith("java.") && !
              name.startsWith("javax.") && !name.startsWith("com.sun.
              ") && !name.startsWith("sun.")) {
23            List<Pattern> patterns = new ArrayList<Pattern>();
24
25            // we will parse the entries each time as we cannot
                  cache this easily
26            // a Cache field cannot be added due to the
                  limitations of the redefine mechanism
27            // we cannot reference non bootclasspath entries (rt.
```

```
                jar) as this is one level lower as the application
                CL
28
29              // premature optimization is the root of all evil. As
                long as this will work fast enough will leave as is
                .
30              Pattern FQN_PATTERN = Pattern.compile("([\\p{L}_$
                \\*][\\p{L}\\p{N}_$\\*]*\\.)*[\\p{L}_$\\*][\\p{L}\\
                p{N}_$\\*]*");
31
32              String[] split = filter.split(",");
33              for (String s : split) {
34                  if (s != null && s.trim().length() > 0) {
35                      if (FQN_PATTERN.matcher(s.trim()).matches()) {
36                          patterns.add(Pattern.compile(s.trim().
                            replace(".", "\\.").replace("*", ".+").
                            replace("$", "\\$")));
37                      } else {
38                          Logger.getGlobal().log(Level.WARNING, "
                            Path Hole Agent has malformed filter
                            entry = " + s);
39                      }
40                  }
41              }
42
43              for (Pattern pattern : patterns) {
44                  if (pattern.matcher(name).matches()) {
45                      throw new ClassNotFoundException(name.replace(
                            ".", "/"));
46                  }
47              }
48          }
49      }
50  }
```

Da pa to naredimo, uporabimo odlično knjižnico za manipulacijo Javine vmesne kode imenovano ASM. Na voljo na naslovu http://asm.ow2.org/.

```
1 private static byte[] prependToClassLoader() {
2     Object[] ourEnhancementsToLoadClass = getMethodNode(PathHole.
          class, "bytecodeToPrepend", "(Ljava/lang/String;Z)V");
3     Object[] jlClassLoader = getMethodNode(ClassLoader.class, "
          loadClass", "(Ljava/lang/String;Z)Ljava/lang/Class;");
4
5     // get the bytecode to prepend
6     InsnList prependInst = ((MethodNode) ourEnhancementsToLoadClass
          [1]).instructions;
7
8     // lets get rid of the return statement
9     // remove the optional label
10    if (prependInst.getLast().getOpcode() < 0) {
11        prependInst.remove(prependInst.getLast());
12    }
13    // remove the return inst. It doesn't take any args so this is
          all we need
14    prependInst.remove(prependInst.getLast());
15
16    // now add this to loadClass method of jlClassLoader
17    InsnList baseInst = ((MethodNode) jlClassLoader[1]).
          instructions;
18    baseInst.insertBefore(baseInst.getFirst(), prependInst);
19
20    ClassNode clClassNode = (ClassNode) jlClassLoader[0];
21    // we just need to add any fields referenced by the prepended
          bytecode to the jlClassLoader
22    // ClassNode prependClassNode = (ClassNode)
          ourEnhancementsToLoadClass[0];
23
24    // write the new bytecode
25    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS |
          ClassWriter.COMPUTE_FRAMES);
26    clClassNode.accept(cw);
27    return cw.toByteArray();
28 }
```

S tem imamo vse dele sestavljanke, da lahko uspešno testiramo odročno nalaganje izvorne kode.

Vse kar ostanje je, da uporabimo path-hole Java agenta pri izvajanju testov. To pa naredimo tako, da dodamo path-hole knjižnico v konfiguracijo SBTja.

Namestimo ga tako.

```
Seq("-javaagent:" + System.getProperty("user.home") + "/.m2/
    repository/com/typesafe/path-hole/1.0/path-hole-1.0.jar")
```

Tu definiramo Java agenta, ki se bo namestil ob zagonu JVM-ja. Kar je še pomembno je, da naš path-hole jar vsebije t.i. manifest file, ki ima slednjo vsebino:

```
Manifest-Version: 1.0
Can-Redefine-Classes: true
Premain-Class: com.typesafe.path_hole.PathHole
```

Vidimo, da manifest zahteva, da smemo spreminjati obstoječe razrede na JVM-ju ter, da naj se namesti naš path hole agent preko svojega razreda. Tu specificiramo polno kvalificirano ime našega razreda.

5.4 Primer testa

Predstavili bomo test `ForwardMultiJvmSpec`. V igri imamo 3 odročne sisteme, pri čemer je na prvem sistemu na voljo razred `Node1Ping`, ki ni na voljo na ostalih dveh. Na tretjem sistemu je na voljo `Node3Pong`, ki ni na voljo na prvih dveh. Prvi sistem pošlje instanco `Node1Ping` na drugi sistem. Tu se mora zgoditi odročno nalaganje razreda `Node1Ping` na drugem sistemu. Potem drugi sistem posreduje (od tu ime testa ang. Forward) sporočilo na tretji sistem. Tretji sistem mora naložiti `Node1Ping` preko prvega sistema. Tretji sistem pa direktno pošlje prvemu sistemu `Node3Pong`. Tako mora priti do nalaganja kode `Node3Pong` na prvem sistemu preko tretjega sistema.

Poglejmo si še kodo.

```
1 package akka.remote.netty.rcl
2
3 import akka.remote.{AkkaRemoteSpec,
      AbstractRemoteActorMultiJvmSpec}
4 import akka.util.Timeout
5 import akka.actor.{ActorRef, Actor, Props}
6 import akka.dispatch.Await
7
8 // Node 1 forwards Node1Ping msg to Node 2 forwards to Node 3 and
      then Node 3 sends Node3 Pong back to Node 1
9 object ForwardMultiJvmSpec extends AbstractRemoteActorMultiJvmSpec
       {
10   override def NrOfNodes = 3
11
12   class ForwardActor(forwardTo: ActorRef) extends Actor with
         Serializable {
13     def receive = {
14       case msg => forwardTo.tell(msg, sender)
15     }
16   }
17
18   class PongActor extends Actor with Serializable {
19     def receive = {
20       case msg => sender ! Node3Pong
21     }
22   }
23
24   val NODE1_PING_FQN = "akka.remote.netty.rcl.
         ForwardMultiJvmSpec$Node1Ping$"
25   // case objects have $ at the end
26   val NODE3_PONG_FQN = "akka.remote.netty.rcl.
         ForwardMultiJvmSpec$Node3Pong$"
27   // case objects have $ at the end
28
29   case object Node1Ping
30   case object Node3Pong
```

```
31
32
33   import com.typesafe.config.ConfigFactory
34
35   override def commonConfig = ConfigFactory.parseString( """
36     akka {
37       loglevel = "WARNING"
38       actor {
39         provider = "akka.remote.RemoteActorRefProvider"
40       }
41       remote.transport = akka.remote.netty.rcl.
             RemoteClassLoadingTransport
42     }""")
43 }
44
45 import ForwardMultiJvmSpec._
46
47 class ForwardMultiJvmNode1 extends AkkaRemoteSpec(nodeConfigs(0))
       {
48
49   import ForwardMultiJvmSpec._
50
51   val nodes = NrOfNodes
52
53   System.setProperty("path_hole.filter", "*Node2*,*Node3*")
54   System.setProperty("path_hole.unfiltered.cls", "akka.remote.
         netty.rcl.RemoteClassLoader")
55
56
57   import akka.util.duration._
58   import akka.pattern.ask
59
60   implicit val timeout = Timeout(20 seconds)
61
62   "___" must {
63     "___" in {
```

```
64        intercept[ClassNotFoundException] {
65          Class.forName(NODE3_PONG_FQN)
66        }
67        barrier("start")
68        val node2EchoActor = system.actorFor("akka://" + akkaSpec(1)
              + "/user/service-forward")
69        Await.result(node2EchoActor ? Node1Ping, timeout.duration).
              asInstanceOf[AnyRef].getClass.getName must equal(
              NODE3_PONG_FQN)
70        barrier("done")
71      }
72    }
73 }
74
75
76 class ForwardMultiJvmNode2 extends AkkaRemoteSpec(nodeConfigs(1))
       {
77
78    import ForwardMultiJvmSpec._
79
80    val nodes = NrOfNodes
81
82    import akka.util.duration._
83
84    System.setProperty("path_hole.filter", "*Node1*,*Node3*")
85    System.setProperty("path_hole.unfiltered.cls", "akka.remote.
          netty.rcl.RemoteClassLoader")
86
87    implicit val timeout = Timeout(20 seconds)
88
89    "___" must {
90      "___" in {
91        intercept[ClassNotFoundException] {
92          Class.forName(NODE1_PING_FQN)
93          Class.forName(NODE3_PONG_FQN)
94        }
```

```
95        barrier("start")
96        val node3PongActor = system.actorFor("akka://" + akkaSpec(2)
              + "/user/service-echo")
97        system.actorOf(Props {
98          new ForwardActor(node3PongActor)
99        }, "service-forward")
100       barrier("done")
101     }
102   }
103 }
104
105 class ForwardMultiJvmNode3 extends AkkaRemoteSpec(nodeConfigs(2))
      {
106
107   import ForwardMultiJvmSpec._
108
109   val nodes = NrOfNodes
110
111   import akka.util.duration._
112
113   implicit val timeout = Timeout(20 seconds)
114
115   System.setProperty("path_hole.filter", "*Node1*,*Node2*")
116   System.setProperty("path_hole.unfiltered.cls", "akka.remote.
          netty.rcl.RemoteClassLoader")
117
118   "___" must {
119     "___" in {
120       intercept[ClassNotFoundException] {
121         Class.forName(NODE1_PING_FQN)
122       }
123       barrier("start")
124       system.actorOf(Props[PongActor], "service-echo")
125       barrier("done")
126     }
127   }
```

```
128 }
```

`AkkaRemoteSpec` ter `AbstractRemoteActorMultiJvmSpec` je del že obstoječe kode znotraj knjižnice Akka. Poskrbi za zagon sistema igralcev, njegovo imenovanje in ostale malenkosti. Kot vidimo se držimo konvencije imenena tako imamo:

1. `ForwardMultiJvmNode1`, ki pošlje `Node1Ping` drugemu sistemu,
2. `ForwardMultiJvmNode2`, ki posreduje `Node1Ping` tretjemu sistemu,
3. `ForwardMultiJvmNode3`, ki pošlje `Node3Pong` prvemu sistemu.

Opazimo tudi uporabo path-hole agenta. Ostali testi so narejeni po istem kopitu kot opisani test. Imamo še:

1. `MsgWithCycleMultiJvmSpec`, ki testira razrede s cikli,
2. `MsgWithFieldsMultiJvmSpec`, ki testira pravilnost fieldov,
3. `MsgWithMethodMultiJvmSpec`, ki testira pravilnost metod,
4. `RemoteActorDeployMultiJvmSpec`, ki testira pravilnost funkcionalnosti odročnega deploya igralca.

V zgodnji verziji implementacije smo imeli tudi test, ki testira pravilnost vrstnega reda sporočil, vendar, ker imamo blokirajočo verzijo implementacije, se nismo odločili za vključitev tega testa. Med drugim zato, ker je vrednost vzdrževanje testa višja, kot je njegova dodatna vrednost.

Poglavje 6

Implementacija

Implementacija na žalost ali na srečo ni enolično določena, imamo pa kar nekaj možnosti pri implementaciji.

Najprej se moramo odločiti kdo začne odročno nalaganje kode. To je lahko ali pošiljatelj sporočila ali pa prejemnik sporočila. Tretja opcija pa je, da uporabimo kombinacijo obeh.

Kot drugo se moramo odločiti, ali bomo blokirali nit izvajanja, ko bomo začeli nalaganje kode.

In na zadnje se moramo odločiti še, koliko razredov bomo prenesli čez žico, ali samo nujno potrebne ali vse? Ali neko pametno kombinacijo?

Naštete možnosti so podane v spodnji tabeli. 6.1. V sledečem podpoglavju opišeme vse naštete možnosti.

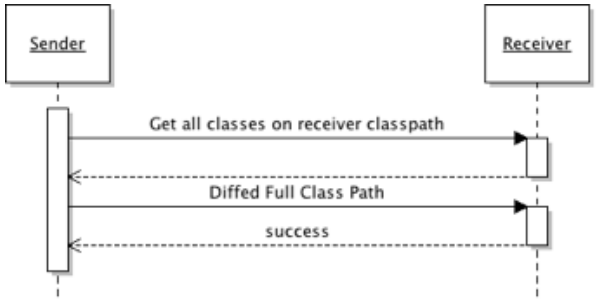
Možnost	Začne	Blokira	Prenese
1	Pošiljatelj	da	vse razrede
2	Pošiljatelj	ne	vse razrede
3	Pošiljatelj	da	samo potrebne razrede
4	Pošiljatelj	ne	samo potrebne razrede
5	Prejemnik	da	samo zahtevan razred
6	Prejemnik	ne	samo zahtevan razred
7	Kombinirano	kombinirano	kombinirano

Tabela 6.1: Tabela odločitev pri implementaciji

6.1 Opis različnih možnosti implementacije

6.1.1 Možnost 1

Preden pošiljatelj pošlje sporočilo, pogleda, ali je z danim odročnim sistemom že komuniciral. Če je že komuniciral, potem samo nadaljuje s pošiljanjem sporočila, v nasprotnem primeru pa začne sinhronizacijo svojega celotnega classpatha z odročnim sistemom. Pri tem blokira nit izvajanja.



Slika 6.1: Komunikacija z odročnim sistemom.

Najprej pošiljatelj od prejemnika zahteva, da mu ta vrne seznam vseh razredov na classpathu vključno z preiskanimi arhivskimi datotekami tipa JAR.

Ko pošiljatelj dobi odgovor, ta na podlagi svojega classpatha naredi razliko. In prejemniku pošlje svoj celoten classpath brez razredov, ki jih prejemnik že

ima.

S tem pristopom smo eksperimentirali in se na koncu za ta pristop nismo odločili. Problem pri tem je, kako dobiti seznam vseh razredov, ki so na voljo.

En način je, da analiziramo t.i. classpath. Gre za seznam direktorijev in JAR datotek na disku, kjer se nahajajo razredi.

```
1 // preko tega ukaza dobimo celoten classpath
2 val cp = System.getProperty("java.class.path")
3 // odstranimo stvari, kot so rt.jar
4 val woHome = filter(cp, System.getProperty("java.home"))
5 // odstranimo še ostale stvari
6 val woLib = filter(woHome, System.getProperty("library.path"))
7
8 // sedaj imamo čisto pot brez sistemskih knjiznic
9 val cleanCp = woLib
```

Problem, ki smo ga imeli s tem je, da zadeva ne deluje, ko izvajamo teste preko SBT build toola. Zato smo se problema lotili tako, da predvidevamo, da se uporablja nalagalnik razredov, ki je podoben razredu `URLClassLoader` oziroma je kar instanca le tega. Preko uporabe refleksije smo uspešno dobili seznam direktorijev in JAR datotek na disku, ki jih je `URLClassLoader` uporabil.

Dani problemi in rešitve nam niso všeč, zato smo opustili nadaljevanje implementacije tega pristopa in poskusili najti bolj elegantno rešitev.

6.1.2 Možnost 2

Identična možnosti z 1, s tem, da ne blokiramo niti izvajanja, kar pomeni, da moramo garantirati vrstni red sporočil tako, da si dokler v celoti ne zaključimo odročnega nalaganja kode, shranjujemo sporočila v predpomnilnik. Ko se odročno nalaganje kode v celoti zaključi, spraznimo predpomnilnik sporočil v pravem vrstnem redu. Pri tem moramo paziti na pravilnosti, saj gre za sočasni sistem.

Ta možnost nam je bila še manj všeč kot predhodna, zato smo nadaljevali z iskanjem.

6.1.3 Možnost 3

Namesto, da pošljemo vse razrede, pošljemo samo razrede, ki jih dano sporočilo potrebuje.

Preden pošljemo sporočilo, preverimo v predpomnilniku, če smo že poslali tak tip sporočila danemu naslovniku. Če smo ga enostavno nadaljujemo. V nasprotnem primeru pa blokiramo nit izvajanja, dokler ne končamo s pošliljanjem vseh potrebih razredov.

Dano sporočilo analiziramo za vse uporabljene razrede. In tudi za razrede, ki jih najdeni razredi uporabljajo. Tako dobimo celoten seznam potrebnih razredov. Potem pošljemo naslovniku naš seznam in on nam odgovori s seznamom brez razredov, ki jih sam že ima. Tako potem pošljemo čez žico samo razrede, ki jih potrebuje naslovnik.

Pri tem pristopu se pojavi robni primer, in sicer, če pošiljateljevo sporočilo vsebuje kodo, kot je recimo `Class.forName("foo.Bar")`.

Z našo analizo vmesne kode ne moremo ugotoviti, da potrebujemo tudi razred `foo.Bar`. Lahko bi izboljšali analizo kode, vendar bi nam to enostavno vzelo preveč časa in tudi dvomimo, da bo naša analiza v celoti našla vse razrede.

Alternativno bi lahko kombinirali ta pristop z možnostjo 5 ali 6. Za ta pristop se tudi nismo odločili. Predvsem zato, kar bi morali imeti še bolj kompleksen predpomnilnik, morali bi zelo paziti na sočasnost in predvsem bi morali imeti zelo kompleksno analizo vsebine razredov za vse uporabljene razrede.

6.1.4 Možnost 4

Identična prejšni možnosti, s to razliko, da ne blokiramo niti izvajanja. S tem moramo imeti še en dodaten predpomnilnik, ki za časa sinhronizacije

razredov pomni sporočila, ki se po končani sinhronizaciji v pravem vrstnem redu pošljejo. Podobno kot razlika med možnostjo 1 in 2.

6.1.5 Možnost 5

Tukaj namesto, da pošiljatelj garantira, da bodo njegovi razredi na voljo na naslovniku, to stori naslovnik. In sicer, ko naslovnikov nalagalec razredov rabi razred, ki ni na voljo, vpraša izvirni sistem razreda za vsebino le-tega. Izvirni sistem je tisti na katerem se nahaja ta razred lokalno. Lahko je enak sistemu pošiljatelja. Lahko je pa pošiljatelj samo posrednik med našim in izvirnim sistemom.

Pri tem blokira nit izvajanja, dokler ne dobi odgovora oziroma dokler ne pride do prekoračitve časa.

Za to možnost smo se na koncu odločili, ker je najbolj enostavna in jo je najlažje robustno sprogramirati.

Prav tako smo dodatno optimizirali nalaganje kode. Ne nalagamo enega razreda na enkrat, ampak naložimo razred in vse njegove direktno uporabljene razrede. S tem precej skrajšamo količino pogovora med pošiljateljem in naslovnikom.

Ker blokiramo nit izvajanja, je nujno, da pogovor o nalaganju razreda poteka po drugi niti. Za to situacijo smo implementirali ločen skupek niti.

6.1.6 Možnost 6

Podobna možnosti 5, vendar s tem, da ne blokiramo niti izvajanja v nalagalniku razredov, vendar moramo loviti `ClassNotFoundException`.

S tem pristopom smo veliko eksperimentirali, vedar se je izkazal za najslabšega, ker smo morali drastično posegati v izvorno kodo.

6.1.7 Možnost 7

Smiselna kombinacija zgoraj naštetih možnosti. S tem nismo eksperimentirali. Vendar se nam zdi, da bi možnost 3 in možnost 5 zelo dobro delovale skupaj.

6.2 Opis izbrane možnosti implementacije

Odločili smo se za možnost 5. In sicer odročno nalaganje razredov začne prejemnik. In sicer prejemnikov nalagalec razredov. Pri tem blokira nit. Dodatno smo se odločili, da predpomnemo še vse razrede, ki jih razred uporablja vendar, jih mi nimamo.

Akka implementira omrežno funkcionalnost v veliki meri s pomočjo knjižnice Netty. Knjižnica Netty je zelo popularna odprtokodna knjižnica za programiranje omrežja v asinhronem stilu. Tako imenovi dogodkovno usmerjen (ang. event driven) stil.

Naša implementacija ne spreminja izvorne kode knjižnice, ampak samo dodaja funkcionalnost. Akka je dokaj razširljiva, zato nam ni bilo treba spreminjati izvorne kode. Med verzijo 1.x in 2.x je velika razlika. Naše delo smo začeli v času verzije 1.x kot projekt Googlovega poletja kode. Verzija 1.x je bila dokaj ne razširljiva in zaradi tega smo morali spremeniti drobovje, zaradi česar tudi naša verzija ni bila vključena v samo knjižnico.

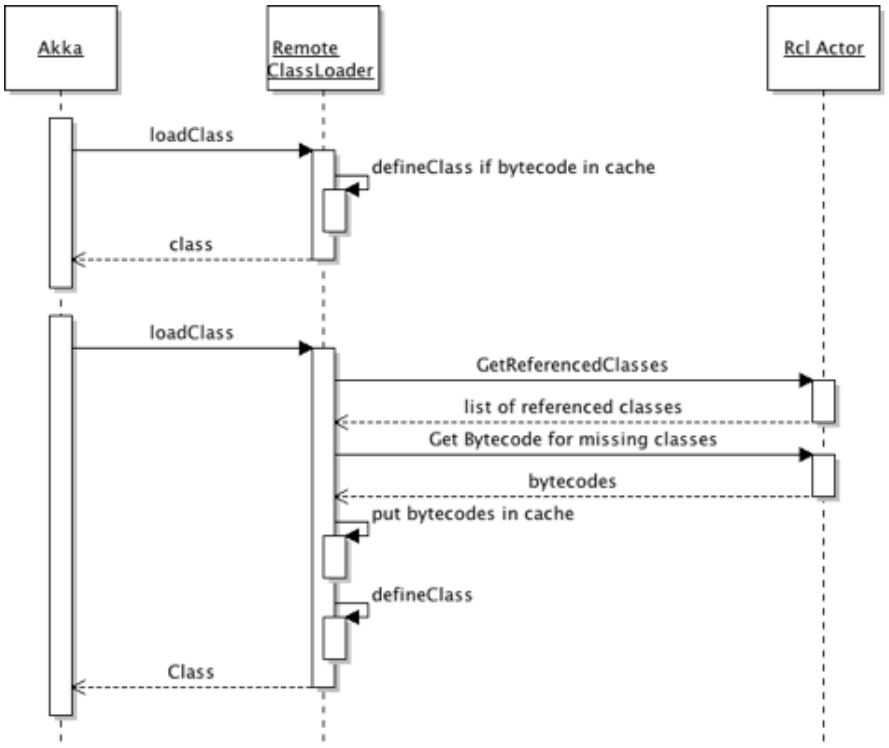
V okviru našega diplomskega dela smo se lotili nove verzije odročnega nalaganja, ki bazira na Akka verziji 2.x. Naša funkcionalnost je v celoti implementirana kot razširitev v obliki RemoteTransporta. V prilogi A smo priložili izvorno kodo le-tega. Uporabimo, ga tako, da v nastavitvah definiramo:

```
remote.transport = akka.remote.netty.rcl.
    RemoteClassLoadingTransport
```

Celotna koda se nahaja na naslovu `https://github.com/avrecko/akka/tree/wip-rcl`

V okviru `RemoteClassLoadingTransport`a hranimo odročne nalagalnike

razredov za vsakega klienta posebej. Bisto implementacije je ravno v teh nalagalnikih razredov. Odročno nalaganje kode začne ravno nalagalnik kode.



Slika 6.2: Potek odročnega nalaganja kode.

Ko naš sistem igralcev (Akka) hoče deserializirati, serializirati instancirati ali kakorkoliže dostopati do razreda, ki ni na voljo v našem sistemu, potem naš nalagalnik kode najprej preveri, če imamo definicijo razreda na voljo v našem predpomnilniku. Če tako, kličemo metodo `defineClass`, ki iz Javine vmesne kode naredi nov razred, in to vrnemo.

V nasprotnem primeru vprašamo odročni sistem, iz katerega vemo, da izhaja zahtevan razred. Vprašamo najprej za seznam vseh razredov, ki jih dani razred direktno uporablja. Nato iz seznama izločimo razrede, ki jih imamo mi lokalno na voljo, nato pa vprašamo odročnega igralca, da naj nam vrne vsebino razredov. Ko nam vrne, shranimo vsebino v predpomnilnik. To bo pospešilo

nalaganje in zmanjšalo čas komunikacije. Nato kličemo `defineClass`. Kot vidimo uporabljamo ideje možnosti 3.

Prednost tega pristopa je, da naša analiza uporabljenih razredov ne rabi biti 100 % zanesljiva. V najslabšem primeru se bo proces odročnega nalaganja razredov začel še za zgrešeni razred.

6.3 Iskanje uporabljenih razredov

Da dobimo seznam vseh direktno uporabljenih razredov, smo najprej uporabili knjižnico ASM. Gre za eno izmed najbolj popularnih knjižnic za analizo Javine vmesne kode.

Koda zgleda tako:

```
1 def getReferences(bytecode: Array[Byte]): java.util.Set[String] =
    {
2     val classNames = new HashSet[String]();
3     val classReader = new ClassReader(bytecode);
4     val emptyVisitor = new EmptyVisitor();
5
6   val remapper = new ClassNameRecordingRemapper(classNames);
7
8   classReader.accept(new RemappingClassAdapter(emptyVisitor,
        remapper), 0);
9   classNames
10 }
11
12 class ClassNameRecordingRemapper(classNames: java.util.Set[String
    ]) extends Remapper {
13
14    override def mapType(resource: String) = {
15      val fqn = resource.replaceAll("/", ".")
16      if (fqn != null && !fqn.startsWith("java.") && !fqn.
          startsWith("scala.")) {
17        classNames.add(fqn)
18      }
```

```
19          resource
20        }
21 }
```

Ker bi pa želeli minimizirati odvisnost od zunanjih knjižnic smo dano funkcionalnost napisali sami. Vsebina razreda ja zapisana v formatu Class. Definira ga JSR-202 [6]. In sicer analiziramo samo vsebino t.i. Constant poola. S tem ne moremo ugotoviti uporabljenih razredov v metodah. Vendar, če zgrešimo kakšen razred to ne predstavlja problema. V tem primeru bo nalagalec razredov samo ponovno začel proces odročnega nalaganja razredov. V prilogi B smo dali izvorno kodo implementacije in testa za občutek.

Poglavje 7

Zaključek

Kot vidimo, sama funkcionalnost ni nekaj posebnega. Je pa sama implementacija zahtevala tehtne premisleke in zadostno mero eksperimentiranja, kako najbolj učinkovito izvesti stvar.

Spoznali smo se z večjem številom različnih knjižnic, orodij in pristopov. Spoznali smo se z:

- SBT (ang. Simple Build Tool),
- GIT (Git Hub),
- Multi-JVM-Tests vtičnik za SBT,
- Path-hole naša Java Agent knjižnica za skrivanje razredov,
- Netty (knjižnica za delo z omrežjem),
- Obstoječa koda Akke,
- Programiranje z ročnim zaklepanjem,
- Detajli nalagalnikov razredov,
- Google Protocol Buffers,

- ASM (analiza bytecode),
- Itd.

Na koncu lahko rečemo, da smo zadovoljni z implementacijo in dejstvom, da zadeva živi v odprotokodnem ekosistemu in bo koristila širšemu krogu ljudi.

Dodatek A

Del izvorne kode implementacije

```
1 package akka.remote.netty.rcl
2
3 import akka.remote.RemoteSettings
4 import akka.remote.netty.NettyRemoteTransport
5 import org.jboss.netty.channel._
6 import org.jboss.netty.handler.execution._
7 import akka.remote.{ RemoteMessage, RemoteActorRefProvider }
8 import com.google.protobuf.ByteString
9 import akka.remote.RemoteProtocol.{ RemoteMessageProtocol,
      AkkaRemoteProtocol }
10 import collection.immutable.HashSet
11 import collection.mutable.HashMap
12 import akka.dispatch.Await
13 import java.net.URL
14 import akka.actor.{ Props, Actor, ActorRef, ExtendedActorSystem }
15 import java.util.concurrent.locks.ReentrantReadWriteLock
16 import akka.event.{ LoggingAdapter, Logging }
17 import scala.collection._
18 import akka.util.NonFatal
19
20 class RemoteClassLoadingTransport(system: ExtendedActorSystem,
      provider: RemoteActorRefProvider) extends NettyRemoteTransport(
      system, provider) {
```

```
21
22  // 1 thread should be plenty
23  val nrOfRclThreads = 1;
24
25  // the RCL stuff has to be processed by a separate thread
        because we might block in the classloader or the user might
        block
26  override def createPipeline(endpoint: => ChannelHandler,
        withTimeout: Boolean): ChannelPipelineFactory = {
27    new ChannelPipelineFactory {
28      def getPipeline = PipelineFactory(PipelineFactory.
            defaultStack(withTimeout).dropRight(1).toSeq ++ Seq(
            sedaRclPriorityHandler, endpoint))
29    }
30  }
31
32  lazy val rclExecutor = new OrderedMemoryAwareThreadPoolExecutor(
33    nrOfRclThreads,
34    settings.MaxChannelMemorySize,
35    settings.MaxTotalMemorySize,
36    settings.ExecutionPoolKeepalive.length,
37    settings.ExecutionPoolKeepalive.unit,
38    system.threadFactory)
39
40  lazy val superExecutor = new
        OrderedMemoryAwareThreadPoolExecutor(
41    settings.ExecutionPoolSize,
42    settings.MaxChannelMemorySize,
43    settings.MaxTotalMemorySize,
44    settings.ExecutionPoolKeepalive.length,
45    settings.ExecutionPoolKeepalive.unit,
46    system.threadFactory)
47
48  lazy val sedaRclPriorityHandler = new ExecutionHandler(new
        ChainedExecutor(new ChannelEventRunnableFilter {
49    def filter(event: ChannelEventRunnable) = {
```

```
50        event.getEvent match {
51          case me: MessageEvent => me.getMessage match {
52            case arp: AkkaRemoteProtocol => RclMetadata.isRclChatMsg
                  (arp)
53            case _                       => false
54          }
55          case _ => false
56        }
57      }
58    }, rclExecutor, superExecutor))
59
60    val systemClassLoader = system.dynamicAccess.classLoader
61
62    val systemClassLoaderChain: HashSet[ClassLoader] = {
63      var clChain = new HashSet[ClassLoader]()
64      var currentCl = systemClassLoader
65      while (currentCl != null) {
66        clChain += currentCl
67        currentCl = currentCl.getParent
68      }
69      clChain
70    }
71
72    // replace dynamic access with our thread local dynamic access
73    val threadLocalDynamicAccess = new
        ThreadLocalReflectiveDynamicAccess(systemClassLoader)
74    ReflectionUtil.setField("_pm", system, threadLocalDynamicAccess)
75
76    lazy val originAddressByteString = ByteString.copyFrom(address.
        toString, "utf-8")
77
78    val remoteClassLoaders: HashMap[ByteString,
        ReflectiveDynamicAccess] = HashMap()
79
80    private val remoteClassLoadersLock = new ReentrantReadWriteLock
81
```

```
82  // this is the actor we query for RCL stuff that is process by
        seperated thread
83  system.actorOf(Props { new RclActor(systemClassLoader) }, "Rcl-
        Service")
84
85  // on received just make sure the "context" has the correct
        classloader set
86  override def receiveMessage(remoteMessage: RemoteMessage) {
87    RclMetadata.getOrigin(remoteMessage) match {
88      case `originAddressByteString` => {
89        threadLocalDynamicAccess.dynamicVariable.withValue(
              systemClassLoader) {
90          super.receiveMessage(remoteMessage)
91        }
92      }
93      case someOrigin: ByteString => {
94        val rcl = getClassLoaderForOrigin(someOrigin)
95
96        threadLocalDynamicAccess.dynamicVariable.withValue(rcl) {
97          super.receiveMessage(remoteMessage)
98        }
99      }
100     case _ => {
101       threadLocalDynamicAccess.dynamicVariable.withValue(
              systemClassLoader) {
102         super.receiveMessage(remoteMessage)
103       }
104     }
105   }
106 }
107
108 def getClassLoaderForOrigin(someOrigin: ByteString):
        RemoteClassLoader = {
109   remoteClassLoadersLock.readLock.lock
110   try {
111     remoteClassLoaders.get(someOrigin) match {
```

```
112         case Some(cl) => cl
113         case None =>
114           remoteClassLoadersLock.readLock.unlock
115           remoteClassLoadersLock.writeLock.lock //Lock upgrade,
                not supported natively
116           try {
117             try {
118               remoteClassLoaders.get(someOrigin) match {
119                 //Recheck for addition, race between upgrades
120                 case Some(cl) => cl //If already populated by
                      other writer
121                 case None => //Populate map
122                   log.debug("Creating new Remote Class Loader with
                        Origin {}.", someOrigin.toStringUtf8)
123                   val cl = new ReflectiveDynamicAccess(new
                      RemoteClassLoader(systemClassLoader, system.
                      actorFor(someOrigin.toStringUtf8 + "/user/Rcl
                      -Service"), someOrigin, log, new
                      RemoteSettings(system.settings.config, system
                      .name)))
124                   remoteClassLoaders += someOrigin -> cl
125                   cl
126               }
127             } finally {
128               remoteClassLoadersLock.readLock.lock
129             } //downgrade
130           } finally {
131             remoteClassLoadersLock.writeLock.unlock
132           }
133       }
134     } finally {
135       remoteClassLoadersLock.readLock().unlock()
136     }
137   }
138
139   // just tag the message with the origin and in case of RCL
```

```
      messages with RCL tag
140   override def createRemoteMessageProtocolBuilder(recipient:
        ActorRef, message: Any, senderOption: Option[ActorRef]) = {
141     val pb = super.createRemoteMessageProtocolBuilder(recipient,
          message, senderOption)
142
143     message match {
144       case rcl: RclChatMsg => RclMetadata.addRclChatTag(pb)
145       case ref: AnyRef => {
146         val name = ref.getClass.getName
147         if (!name.startsWith("java.") && !name.startsWith("scala."
              )) {
148           ref.getClass.getClassLoader match {
149             case rcl: RemoteClassLoader => {
150               RclMetadata.addOrigin(pb, rcl.originAddress)
151             }
152             case cl: ClassLoader => systemClassLoaderChain(cl)
                    match {
153               case true => RclMetadata.addOrigin(pb,
                      originAddressByteString)
154               case _    => log.warning("Remote Class Loading does
                      not support sending messages loaded outside Actor
                       System's classloader.\n{}", message)
155             }
156             case null => // null ClassLoader e.g. java.lang.String
                    this is fine
157           }
158         }
159       }
160
161       case _ => // not AnyRef even less of a problem
162     }
163     pb
164   }
165 }
166
```

```
167 import akka.util.Timeout
168 import akka.util.duration._
169 import akka.pattern.ask
170
171 class RemoteClassLoader(parent: ClassLoader, origin: ActorRef, val
        originAddress: ByteString, log: LoggingAdapter, settings:
      RemoteSettings) extends ClassLoader(parent) {
172
173   implicit val timeout = Timeout(settings.
        RemoteSystemDaemonAckTimeout)
174
175   val preloadedClasses = new mutable.HashMap[String, Array[Byte
        ]]()
176
177   var innerCall = false
178
179   // normally it is not possible to block in here as this will in
        fact block the netty dispatcher i.e. no new stuff on this
        channel
180   // but we are using a special thread just for this so this is
        safe
181   override def findClass(fqn: String): Class[_] = {
182     if (innerCall) throw new ClassNotFoundException(fqn)
183
184     log.debug("ClassLoader#findClass({}) from {}.", fqn, origin.
          path.address)
185
186     preloadedClasses remove (fqn) orNull match {
187       case null => doRcl(fqn)
188       case bytecode: Array[Byte] => {
189         log.debug("Bytecode for {} found in preloaded cache from
              {}.", fqn, origin.path.address)
190         defineClass(fqn, bytecode, 0, bytecode.length)
191       }
192     }
193   }
```

```
194
195   def doRcl(fqn: String): Class[_] = {
196     try {
197       log.debug("Initiated RCL for {} from {}.", fqn, origin.path.
            address)
198       val referencedClasses = Await.result(origin ?
            GiveMeReferencedClassesOf(fqn), timeout.duration) match {
199         case r: ReferencedClassesOf => r.refs
200         case _ => {
201           // if we can't get references we are sure we cannot get
                the bytecode or is corrupt
202           log.warning("Failed to find referenced classes for {}.")
203           throw new ClassNotFoundException(fqn)
204         }
205       }
206       log.debug("Got list of referenced classes for {} from {}.\n
            {}", fqn, origin.path.address, referencedClasses
            deepToString)
207       val withoutAlreadyAvailable = referencedClasses filter (!
            isAlreadyLoaded(_)) filter (_ != fqn)
208       log.debug("Requesting only the following bytecodes from {}.\
            n{}", origin.path.address, withoutAlreadyAvailable
            deepToString)
209
210       val bytecodes = Await.result(origin ? GiveMeByteCodesFor(fqn
            , withoutAlreadyAvailable), timeout.duration).
            asInstanceOf[ByteCodesFor]
211
212       log.debug("Got bytecodes from {}.\n{}", origin.path.address,
             bytecodes.entries deepToString)
213
214       bytecodes.entries foreach (_ match {
215         case ByteCodeFor(fqn, bytecode) => preloadedClasses += fqn
              -> bytecode
216         case _                          => log.error("Bug in RCL
              code. ByteCodesFor contained invalid entries.")
```

```
217         })
218         val bytecode = bytecodes.first.bytecode
219         defineClass(fqn, bytecode, 0, bytecode.length)
220       } catch {
221         case e: Exception => {
222           log.debug("Failed to get requested bytecode for class {}
                from {}.\n{}", fqn, origin.path.address, e)
223           throw new ClassNotFoundException(fqn)
224         }
225       }
226     }
227
228     def isAlreadyLoaded(fqn: String): Boolean = {
229       try {
230         innerCall = true
231         loadClass(fqn)
232         return true
233       } catch {
234         case NonFatal(_) => false
235       } finally {
236         innerCall = false
237       }
238     }
239 }
240
241 object RclMetadata {
242
243     def isRclChatMsg(arp: AkkaRemoteProtocol): Boolean = {
244       if (arp.hasInstruction) false
245       import scala.collection.JavaConversions._
246       arp.getMessage.getMetadataList.collectFirst({
247         case entry if entry.getKey == "rclChatMsg" => true
248       }).getOrElse(false)
249     }
250
251     def addRclChatTag(pb: RemoteMessageProtocol.Builder) {
```

```
252     val metadataBuilder = pb.addMetadataBuilder()
253     metadataBuilder.setKey("rclChatMsg")
254     metadataBuilder.setValue(ByteString.EMPTY)
255   }
256
257   def addOrigin(pb: RemoteMessageProtocol.Builder, origin:
          ByteString) {
258     val metadataBuilder = pb.addMetadataBuilder()
259     metadataBuilder.setKey("origin")
260     metadataBuilder.setValue(origin)
261   }
262
263   def getOrigin(rm: RemoteMessage): ByteString = {
264     val rmp: RemoteMessageProtocol = rm.input
265     import scala.collection.JavaConversions._
266     rmp.getMetadataList.collectFirst({
267       case entry if entry.getKey == "origin" => entry.getValue
268     }).orNull
269   }
270
271 }
272
273 class RclActor(val cl: ClassLoader) extends Actor {
274
275   val log = Logging(context.system, this)
276
277   def receive = {
278     case GiveMeByteCodeFor(fqn) => {
279       try {
280         log.debug("Recieved bytecode request for {} from {}.", fqn
              , sender.path.address)
281         sender ! ByteCodeFor(fqn, getBytecode(fqn))
282       } catch {
283         case NonFatal(_) =>
284           log.warning("Cannot find bytecode for {}.", fqn)
285           sender ! ByteCodeNotAvailable(fqn)
```

```
286
287       }
288     }
289
290     case GiveMeByteCodesFor(fqn, fqns) => {
291       try {
292         log.debug("Recieved bytecodes request for {} from {}.",
              fqns, sender.path.address)
293         sender ! ByteCodesFor(ByteCodeFor(fqn, getBytecode(fqn)),
              fqns.map((fqn) => ByteCodeFor(fqn, getBytecode(fqn)))
              toArray)
294       } catch {
295         case NonFatal(_) =>
296           log.warning("Cannot find bytecode for {}.", fqns)
297           sender ! ByteCodeNotAvailable("")
298
299       }
300     }
301
302     case GiveMeReferencedClassesOf(fqn) => {
303       log.debug("Recieved request for all references of {} from
            {}.", fqn, sender.path.address)
304       try {
305         // this should always unless we don't have the class or is
              corrupt
306         // but we consider if is corrupt that the bytecode is not
              available
307         sender ! ReferencedClassesOf(fqn, ByteCodeInspector.
              findReferencedClassesFor(cl.loadClass(fqn)) toArray)
308       } catch {
309         case NonFatal(_) =>
310           log.warning("Cannot find referenced classes of of {}.",
              fqn)
311
312           sender ! ByteCodeNotAvailable(fqn)
313       }
```

```
314
315       }
316
317       case _ => log.warning("RCL actor received unknown message.
              Might indicate a bug present.")
318     }
319
320     def getBytecode(fqn: String): Array[Byte] = {
321       val resourceName = fqn.replaceAll("\\.", "/") + ".class"
322       cl.getResource(resourceName) match {
323         case url: URL => IOUtil.toByteArray(url)
324         case _        => throw new ClassNotFoundException(fqn)
325       }
326     }
327
328 }
329
330 sealed trait RclChatMsg
331
332 // RCL Questions
333 case class GiveMeByteCodeFor(fqn: String) extends RclChatMsg
334 case class GiveMeByteCodesFor(fqn: String, fqns: Array[String])
        extends RclChatMsg
335 case class GiveMeReferencedClassesOf(fqn: String) extends
        RclChatMsg
336
337 // RCL Answers
338 case class ByteCodeFor(fqn: String, bytecode: Array[Byte]) extends
         RclChatMsg
339 case class ByteCodesFor(first: ByteCodeFor, entries: Array[
        ByteCodeFor]) extends RclChatMsg
340 case class ReferencedClassesOf(fromFqn: String, refs: Array[String
        ]) extends RclChatMsg
341
342 case class ByteCodeNotAvailable(fqn: String) extends RclChatMsg
```

Dodatek B

Izvorna koda iskanja uporabljenih razredov

```
1 package akka.remote.netty.rcl
2
3 import java.io.DataInputStream;
4
5 object ByteCodeInspector {
6
7   val magic = 0xCAFEBABE
8
9   val utf8_tag = 1
10  val int_tag = 3
11  val float_tag = 4
12  val long_tag = 5
13  val double_tag = 6
14  val class_ref_tag = 7
15  val string_ref_tag = 8
16  val field_ref_tag = 9
17  val method_ref_tag = 10
18  val intfce_ref_tag = 11
19  val name_type_desc_tag = 12
20
```

```
21  def findReferencedClassesFor(klass: Class[_]): List[String] = {
22    // we will analyse just the constant pool no need to load the
          whole file
23    val resource = klass.getClassLoader.getResource(klass.getName.
          replace('.', '/') + ".class");
24    val input = new DataInputStream(resource.openStream());
25
26    try {
27
28      input.readInt match {
29        case 0xCAFEBABE => // good
30        case _          => throw new RuntimeException("Not a
              bytecode file.")
31      }
32
33      input.readUnsignedShort(); // minor
34      input.readUnsignedShort(); // major
35
36      // this values is equal to the number entries in the
            constants pool + 1
37      val constantPoolEntries = input.readUnsignedShort() - 1;
38
39      // we will fill this Map with UTF8 tags
40      val utfTags = scala.collection.mutable.HashMap[Int, String
            ]()
41
42      // we will mark which utf8 tags point to class and to
            name_and_type
43      val classTags = scala.collection.mutable.ArrayBuffer[Int]()
44      val descTags = scala.collection.mutable.ArrayBuffer[Int]()
45
46      // loop over all entries in the constants pool
47      var i = 0
48      while (i < constantPoolEntries) {
49        i += 1
50        // the tag to identify the record type
```

```
51          input.readUnsignedByte() match {
52            case 1 => utfTags += i -> input.readUTF
53            case 7 => classTags += input.readUnsignedShort
54            case 12 => {
55              input.readUnsignedShort // don't care about the name
56              descTags += input.readUnsignedShort
57            }
58            case 3 | 4 | 9 | 10 | 11 => {
59              // this tags take 4 bytes and we don't care about them
60              input.readInt
61            }
62            case 5 | 6 => {
63              input.readLong // this take 8 bytes
64              i += 1 // entry takes 2 slots
65            }
66            case 8 => input.readUnsignedShort // and this 2 bytes
67            case _ => throw new RuntimeException("Encountered unknow
                  constant pool tag. Corrupt bytecode or new format.")
68          }
69        }
70
71        val answer = scala.collection.mutable.HashSet[String]()
72
73        // read the utf8 tags for fqn class names
74        classTags.foreach { answer += utfTags(_) replaceAll ("/", ".
             ") }
75
76        descTags.foreach {
77          answer ++= """L[^;]+;""".r findAllIn utfTags(_) map {
78            (s: String) => s drop (1) dropRight (1) replaceAll ("/",
                  ".")
79          }
80        }
81
82        answer.toList
83      } finally {
```

```
84          input.close
85        }
86    }
87 }
```

```
1 package akka.remote.netty.rcl
2
3 import org.scalatest.FlatSpec
4 import org.scalatest.matchers._
5
6 class ByteCodeInspectorSpec extends FlatSpec with ShouldMatchers {
7
8   "ByteCodeInspector" should "be able to list fqn of *most* of the
        classes the given class references" in {
9     val refs = ByteCodeInspector.findReferencedClassesFor(classOf[
        ToInspect]) toSet
10
11    refs should contain("java.math.BigDecimal") // fields
12    refs should contain("java.util.Calendar") // referenced
        methods
13    refs should contain("java.util.TimeZone") // and
14    refs should contain("java.util.Locale") // params
15    refs should contain("java.util.ArrayList") // params
16
17    // should contain itself
18    refs should contain("akka.remote.netty.rcl.
        ByteCodeInspectorSpec$ToInspect")
19
20    // this is where *most* comes from, constant pool does not
        include methods the class defines
21    // it makes sense as this information is available in the
        method pool
22    // we are not parsing the method pool therefore we lack to get
         the references from our method definitions
23    // but the same class could be referenced elswhere
24    refs should not contain ("java.lang.String")
25    refs should not contain ("java.util.Date")
26  }
27
28  class ToInspect() {
29
```

```
30      val field: java.math.BigDecimal = null
31
32      def methodParamsRefsNotAvailable(a: String) {
33        java.util.Calendar.getInstance(null /*TimeZone*/ , null /*
              Locale*/ )
34      }
35
36      def returningNotAvailable(): java.util.Date = null
37
38      def referenced() {
39        val available = new java.util.ArrayList()
40      }
41    }
42
43 }
```

Slike

Tabele

Literatura

[1] J. Gosling, B. Joy, G. Steele, G. Bracha. "Java™ Language Specification, The (3rd Edition)", Addison Wesley, 2005.

[2] J. Bloch. "Effective Java (2nd Edition)", Addison Wesley, 2008. 260-262

[3] C. Hewitt, P. Bishop, R. Steiger. "A universal modular ACTOR formalism for artificial intelligence", *Proceedings of the* IJCAI'73, 235-245

[4] (2012) Multi-JVM Testing. Dostopno na: http://doc.akka.io/docs/akka/2.0.2/dev/multi-jvm-testing.html

[5] (2012) JSR 87: Java™ Agent Services. Dostopno na: http://jcp.org/en/jsr/detail?id=087

[6] (2012) JSR 202: Java™ Class File Specification Update. Dostopno na: http://jcp.org/en/jsr/detail?id=202