# Developing the Scala Bindings to the Fly Object Space

Channing Walton
Casual Miracles Ltd
Channing.Walton@casualmiracles.com

Nigel Warren
Zink Digital Ltd
Brunel University
Nigel.Warren@zink-digital.com

## Abstract

Fly is a lightweight Object Space that can distribute and coordinate information on clusters of computers in the form of Objects. It follows the form of Tuple Spaces in Linda and Java Spaces in Jini. Fly is language independent and hence can be bound into various programming languages via the development of specific language stubs that expose the interface to the Space to client programs.

In this paper we contrast and compare the interface of the Java Bindings with the development and design of the Scala bindings and interfaces, with particular reference to the introduction of Scala language specific features such as the use of Options and the Scala Actors model.

***Categories and Subject Descriptors*** D.3.3 **Programming Languages**: Design – Grid computing, LINDA, Java, Scala, Actors.

***General Terms*** Algorithms, Performance, Design, Reliability, Experimentation, Languages.

***Keywords*** *Object Spaces; Distributed Computing, Parallel Computing, Programming, Actors, generative communication*.

## 1. Background

Tuple Spaces formed the theoretical underpinning of the Linda Programming Language and Environment proposed by David Gelernter and Nicholas Carriero [1] in which two or more processes communicate by generating Tuples (data) and sharing these Tuples via a Tuple Space; so called generative programming [2].

The design of Jini's JavaSpaces [3] was heavily influenced by Linda systems but added features to support object orientation such as a richer type system in which all Space Entries were instances of Java Objects [4]; and leasing to support distributed garbage collection patterns. JavaSpaces also removed the 'eval' operation, citing the issues of fairness and security. This left three principle Space operations :-


*SIGPLAN'05* June 12–15, 2005, Location, State, Country.


Write – Write an Entry into the Space for a given lease time (equivalent to the Linda 'in' operation)

Read – Read a copy of an entry from a space by the associative matching of a template object leaving a copy visible to other processes (equivalent to the Linda 'rd' operation)

Take – Remove the entry from the Space by the associative matching of a template object. (equivalent to the Linda 'out' operation)

These operations are all 'blocking' operations performed atomically on the Space implementation and are the primary means of interaction with the Space. However, an asynchronous notification system is also provided in which a client can specify a template object and a method to be performed, should an entry be written to, or taken from the Space that matches the given template.

## 2. An Object Space

Fly retains the requirements for rich types and leases, but drops the dependency on a specific language coding and semantics. The Fly Space server is language neutral and relies on the definition of a lightweight binary interface that specifies the encoding of Space operations and data at the byte (octet) level. To date, language bindings have been developed for Java, Ruby and Scala.

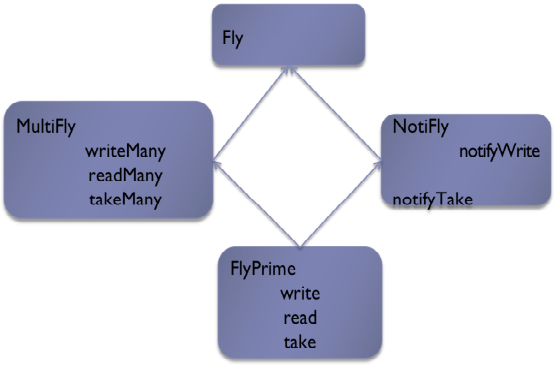


**Figure 1.** Basic Fly Classes and their relationships

The operations available on the Space are split into small groups of operations under simple interfaces that cover the basic Object Space operations described above. These are named FlyPrime; the Notification interfaces are named NotiFly (sic); and the methods used for performing operations on collections of objects are named MultiFly. The relationship between these classes is shown in Figure 1. In the following section we describe the initial port of the Fly Java interface and bindings to the Scala [5] version and introduce Scala language idioms as they become appropriate in the implementation.

## 3. Initial Scala Port

The Scala port of the Java client library was carried out in two steps: a syntactic port where Java source was translated line by line to Scala, followed by a green-field implementation that draws upon specific Scala idioms not present in Java.

### 3.1.1 Syntactic Port

The first step taken to implement the Scala client library was to systematically transform the Java library to Scala by mapping classes and interfaces to Scala classes and traits. Apart from some minor considerations, such as Java statics being placed in Scala objects, this process is very mechanical and could possibly be automated. However in general we would not advocate blindly automating the transformation in this fashion, as the design of a system in Java, using purely object oriented techniques, would not make the best use of functional programming techniques available in a language like Scala. However, we did take this route for FlyScala as it served as a way in to learning Scala.

For example, this is a FlyJava interface:

```
public interface NotiFly extends FlyPrime {

  boolean notifyWrite(Object template, NotifyHan-
dler handler, long leaseTime);

  boolean notifyTake(Object template, NotifyHan-
dler handler, long leaseTime);

}
```

and here is the equivalent in Scala:

```
trait NotiFly extends FlyPrime {

  def notifyWrite(template: AnyRef, handler: No-
tifyHandler, leaseTime: Long): Boolean

  def notifyTake(template:AnyRef, han-
dler:NotifyHandler, leaseTime:Long):Boolean

}
```

Or a class:

```
public static void main(String[] args) throws Ex-
ception {

  FileInputStream f = new FileInputStream(
                       new File(args[0]));
  DataInputStream dis = new DataInputStream(f);

  StatsDecoder decoder = new StatsDecoder();

  long time = dis.readLong();
  while ( true ) {
     int size = dis.readInt();
     StatsBean [] beans=
            decoder.getStatsArray(dis);
     Stats.writeStats(beans);
     System.out.println("--------------");
     long nextTime = dis.readLong();
     Thread.sleep(nextTime-time);
     time = nextTime;
  }
}
```

and the Scala version:

```
def main(args: Array[String]) {

    val f = new FileInputStream(new
File(args(0)));
    val dis = new DataInputStream(f);

    val decoder = new StatsDecoder();

    var time = dis.readLong();
    while (true) {
      val size = dis.readInt();
      val beans = decoder.getStats(dis);
      StatsPrinter.writeStats(beans);
      System.out.println("--------------");
      val nextTime = dis.readLong();
      Thread.sleep(nextTime - time);
      time = nextTime;
    }
  }
```

Having completed this transformation, the result was a working Scala version of the library albeit with limited value, since a Scala client would not gain anything over the Java version.

### 3.2 Introducing Scala Idioms

#### 3.2.1 Options

It is idiomatic Java to use null to represent something that does not exist. The use of this idiom has been the cause of many wasted hours searching for the source of NullPointerExceptions. Furthermore, there is no hint that a class's field can be null which makes it difficult to reason about the code that one is reading.

To avoid these problems it is possible to introduce a so-called Optional Type, which supports both the existence of a value and no value. Thus readers of the code will understand immediately that the field is optional, but without the runtime NPE issues.

Scala has a library class called Option that represents an optional value which has two subclasses: Some and None. Thus reading from an object space, FlyJava will return null when nothing matches a query template. FlyScala returns an Option:

```
trait FlyPrime {

   def read[T <: AnyRef](template: T, waitTime:
Long): Option[T]

   }
```

It can be used as follows in the classic distributed Ping Pong application.

```
fly.read(template, 0L) match {
  case None => println("No ball in play")
   case Some(gameBall) =>
     println("Received ball - game on!")
     …
   }
```

Or alternatively:

```
fly.read(template,0L).map((x:T)=>doSomething(x))
```

which in the case of Some(x) will result in a new Some(f(x)), or None if the returned Option is None.

Options can be implemented in Java but this is unwieldy because Java does not support first class functions or pattern matching.

#### 3.2.2 Return Values

The next step in the port was to make use of returned values from blocks such as if-else and try-catch. So this expression in Java:

```
int iterations = 10000;
if(args.length > 0) iterations =
        Integer.parseInt(args[0]);
```

becomes:

```
val iterations = if (args.length > 0)
args(0).toInt else 10000
```

Apart from terseness, the variable iterations, which is mutable and reassigned in the Java version, has been replaced by a constant in the Scala version. Whilst this is a trivial example it does touch upon the important area of immutability and the ability to reason about software which is so important for working with complex systems.

#### 3.2.3 Collections

Below is a common idiom in Java that appears in FlyJava (and many other codebases) to filter a collection:

```
public Collection<FlyServerRep>
        getMatchingReps(String [] tags) {
           Collection matched =
                new ArrayList<FlyServerRep>();
           for (FlyServerRep rep : reps.values()){
               if (rep.tagsMatch(tags)) {
                       matched.add(rep);
               }
           }
           return matched;
       }
```

In Scala this reduces to:

```
def getMatchingReps (tags:Array[String]) =
reps.values.filter(_.tagsMatch(tags)).toList
```

The expression _.tagsMatch(tags) is shorthand for a filter function that accepts an item in the collection and returns a Boolean. We find this much easier to understand and hence reason about; importantly, resulting in the expression of the operation at a higher level of abstraction.

#### 3.2.4 Control Abstraction

Scala enables developers to write code which almost looks like it is part of the language and effectively creates new control structures. An example in FlyJava is code that times how long N iterations of an operation takes:

```
System.out.println("Processing "+itrs+" writes
and reads");
long start = System.currentTimeMillis();
for (int i = 0; i < itrs; i++) {
     space.write(object, 1000);
     space.read(template, 0L);
}
long end = System.currentTimeMillis();
float seconds = (end - start) / 1000.0f;
System.out.println("Taking "+seconds+" seconds");
```

It is obviously possible to extract the block of code being timed by introducing a class instantiated with a Runnable instance, etc. It would be preferable to write this:

```
Time("a message", iterations) {
  space.write(obj, 1000)
  space.take(template, 0L)
}
```

Below is the Scala code that implements “Time”

```
object Time {
  def apply(name: String, iterations: Int)(block:
=> Unit): Unit = {
    println(name)
    val start = System.currentTimeMillis()
    for (i <- 0 until iterations) block
    val end = System.currentTimeMillis()
    val seconds = (end - start) / 1000.0F
    println("Took "+seconds+" seconds")
  }
}
```

Time is a singleton object with an apply method. The apply method means code can call this method without explicitly naming it as in the example above. Note that Time(...) is syntactic sugar for Time.apply(...).

The last parameter list of the apply method takes a function that returns nothing (Unit) which means that the last parameter can be passed as a block.

#### 3.2.5 For Comprehensions

For comprehension are an extremely powerful iteration, filtering and mapping mechanism that can help to reduce the amount of unnecessary code.

For example, the following Java:

```
public StatsBean[] getStatsArray(DataInputStream
dis) throws IOException {
     long statsCount = dis.readLong();
                    StatsBean [] stats = new
               StatsBean[(int)statsCount];
     for (int i = 0; i < statsCount; i++) {
         stats[i] =
StatsBean.makeBeanFromStream(dis);
     }
     return stats;
}
```

is replaced by a relatively simple use of a 'for comprehension'

```
def getStats(dis: DataInputStream):
Seq[StatsBean]
  = for (i <- 0 until dis.readLong().toInt)
       yield StatsBean.makeBeanFromStream(dis)
```

#### 3.2.6 Fold

Another common functional pattern is the use of fold to process lists. In FlyJava we have:

```
public long writeMany(Collection entries, long
lease) {
    long lastLease = 0;
    for (Object entry : entries) {
        lastLease = codec.write( entry, lease );
    }
    return lastLease;
}
```

which can be replaced with the following in Scala:

```
def writeMany(entries: Collection[AnyRef],
               lease: Long): Long
       = (0L /: entries){(previousLease,
               nextEntry) =>
       codec.write(nextEntry, lease)}
```

#### 3.2.7 Scala Friendly API

Fly recommends a standard API for client libraries for consistency and common understanding across different client implementations.

The Notifiable interface enables handlers to be notified when objects matching a template are written to the space. In Java, Notiflable looks like this:

```
Boolean notifyWrite(Object template, Notifiable
handler, Long leaseTime);
```

Because Scala supports blocks and multiple argument lists, the Scala version of Notifiable has an additional method:

```
def notifyWrite(template: AnyRef, leaseTime:
Long)(block: => Unit): Boolean
```

which is idiomatic Scala as well as making client code simpler:

```
fly.notifyWrite(template, 0L) {
       // block gets executed when notification
       // is raised
}
```

This removes the requirement in Java to implement both the implementing class for the Notifiable interface and its notify method.

#### 3.2.8 Functions

Scala is a functional language enabling functions to be passed as arguments to other functions which enables the composition of complex behaviour by assembling functions with other functions. In fact, class methods in Scala can be passed as arguments to other methods and functions.

This technique is used by FlyScala to eliminate duplication without a need for new class hierarchies that add complexity.

#### 3.2.9 Summary

After a syntactic mapping from Java to Scala and the introduction of Scala idioms, FlyScala presented a Scala friendly API to clients. This resulted in a reduction in the number of lines of code of about a third. There being approximately 4000 line of java and approximately 2800 lines of Scala. We provide this metric with all the usual caveats.

As expected, there was no significant change in performance since the design of FlyScala and the work it does is largely the same as FlyJava. Many of the underlying libraries used by FlyScala, particularly IO libraries, are standard Java libraries, and overall performance of Fly is determined for the most part by the underlying hardware and network infrastructure.

## 4. Actors

### 4.1 Introduction

Actors can be used to raise the level of abstraction of concurrent systems. Instead of working with synchronization primitives such

as mutex locks, Actors encapsulate state and behaviour in an object that communicates with other actors via an ordered message queue, or mailbox. Actors were introduced by Carl Hewitt [6] and used famously in Erlang in highly concurrent system.

#### 4.1.1 Actors in FlyScala

One of the contracts for clients of API's like FlyJava is that any callbacks from the server should be dealt with swiftly and the callback thread handed back to the library as soon as possible. This is the case with FlyJava's notification mechanism.

FlyScala took a different approach and made use of Actors in the notification mechanism. Therefore, notifications from the server are handled by sending messages to actors which return the server thread immediately. The actor will handle the notification when resources allow it to do so.

FlyScala offers three methods for notifications of writes and takes (where XXX can be Write or Take)

```
def notifyXXX(template: AnyRef,
              handler: Notifiable,
              leaseTime: Long): Boolean
def notifyXXX(template: AnyRef,
       leaseTime: Long)(block: => Unit): Boolean
def notifyXXX(template: AnyRef,
       leaseTime: Long, actor: Actor): Boolean
```

However, FlyScala implements the first two by wrapping the Notifiable or block in an actor and delegating to the actor based mechanism.

The use of actors in FlyScala has simplified parts of FlyScala's implementation since remote callbacks from the server can be sent immediately to the appropriate actor using the actor's built-in messaging support, rather than having to write custom code.

## 5. Future Work

The port from FlyJava to FlyScala was a learning exercise with the result of providing a more friendly and idiomatic API to Scala users.

However, the basic design of FlyScala is identical to FlyJava and as such does not make use of Scala idioms at a design level. Doing so will almost certainly make FlyScala smaller and simpler. The current version of FlyScala targets Scala 2.7 however in subsequent releases we intend to target 2.8.

We have started work on project called 'Wide' which enables the implementation of a process Trellis architecture [7] for realtime monitoring of streams of data (CEP Streams) which will be implemented Scala using Fly.

Acknowledgments

Thanks to Simon Kent and Miles Sabin.

## References

[1] "How to Write Parallel Programs: A Guide to the Perplexed," Nicholas Carriero and David Gelernter, ACM Computing Surveys, Sept., 1989.

[2] Gelernter, David. "Generative communication in Linda". ACM Transactions on Programming Languages and Systems, volume 7, number 1, January 1985

[3] Eric Freeman, Susanne Hupfer, Ken Arnold: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Professional, 1. June 1999, ISBN 0-201-30955-6

[4] James Gosling, Bill Joy, Guy Steele , Gilad Bracha : The Java Language Specification, The (3rd Edition) Addison-Wesley Professional, 14 June 2000, ISBN 978-0321246783

[5] The Scala Language Specification Version 2.7 - Martin Odersky - Programming Methods Lab - EPFL

[6] Carl Hewitt, Peter Bishop, Richard Steiger: A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI 1973: 235-245

[7] Michael Factor : The process trellis architecture for real-time monitors. – In : Principles and Practice of Parallel Programming – Proceedings of the second ACM SIGPLAN symposium on the Principles and Practice of Parallel Programming. Pages 147 -155 - 1990 – ISBN:0-89791-350-7

.