Building Distributed Systems in Scala with OpenSplice DDS

Angelo Corsaro, Ph.D.
Chief Technology Officer
PrismTech
OMG DDS SIG Co-Chair
angelo.corsaro@prismtech.com

OpenSplice DDS
Delivering Performance, Openness, and Freedom

TYPE MATTERS
The Type Systems Debate

- The debate around type Programming Languages Type Systems has animated computer scientist over several decades and is far from being resolved.

- New Programming Languages are continuing to appear on both sides of the fence.

**Statically Typed (ex.)**

- Scala
- Caml
- F#

**Dynamically Typed (ex.)**

- Erlang
- Groovy
- Ruby
Are you a Type-Phobic Type?

- Proponents of Dynamically Typed Programming Languages advocate against strong typing in favor of reduced verbosity and added flexibility

Yet...Are they so sure that a type system always gets between you and what you are trying to achieve?
Strongly-Typed Type-Systems

- Well designed type-systems don’t add unnecessary verbosity to your application since they use sophisticated inference to derive types.
- Allow the detection of many errors at compile time, thus improving productivity, code quality and reducing the potential for run-time errors.
- Enable the generation of more efficient code.
OpenSplice | DDS

Delivering Performance, Openness, and Freedom


Type Matters in Distributed Systems
Type Systems and Distributed Systems

- The Type System debate has not spared distributed systems
- Today we can find different technologies taking different approaches: from completely untyped to strongly typed
- The differences are in this case more complex to organize:
  - Some technologies support only one type, being a string or an array of bytes (e.g. AMQP)
  - Other technologies support some predefined types (e.g JMS)
  - Other technologies allow the definition of user-specified types (e.g DDS)
Types and Distributed Systems

User Defined Types

T1  T2  T3  T4
Types and Distributed Systems

Pub/Sub Types
- octet sequence

User Defined Types
- T1, T2, T3, T4

AMQP

User Defined Types
- T1, T2, T3, T4

Pub/Sub Types
- T3, T2

User Defined Types
- T1, T2

Pub/Sub Types
- T3, T4

User Defined Types
- T1, T2

Pub/Sub Types
- T1, T2

User Defined Types
- T1, T2
Types and Distributed Systems

JMS

Pub/Sub Types

ObjectMessage

TextMessage

StreamMessage

MapMessage

ByteMessage
Types and Distributed Systems

Pub/Sub Types == User Types!

DDS

T1 T2
T3

T1 T2
T3

T1
T3
T4

T1 T2
T3

T1
T2

OpenSplice DDS

Delivering Performance, Openness, and Freedom


MY MANIFESTO
My Type-System Manifesto

- Modern Statically typed programming languages are very valuable for building **scalable**, **safe**, and **efficient** applications.

- Modern Statically typed middleware systems are very valuable for building **scalable**, **safe**, and **efficient** distributed applications.
Thus

Scala + OpenSplice DDS =

The best choice for building safe, efficient, and scalable distributed systems
OpenSplice DDS
Delivering Performance, Openness, and Freedom

STEPPING INTO SCALA
What is Scala

Scala (pronounced Skah-lah) stands for “**Scalable language**”

- It is a language that carefully and creatively blends Object Oriented and Functional constructs into a statically typed language with sophisticated type inference.

- Scala targets the JVM and .NET runtime and is 100% compatible with Java.
Why should you care?

- Scala is simple to write, extremely compact and easy to understand
- Scala is strongly typed with a structural type system
- Scala uses a sophisticated type inference to deduce types whenever possible
- Scala is an extensible language (many constructs are built in the standard library)
- Scala makes it easy to design Domain Specific Language

Complex Numbers

To explore some of the nice features of Scala, let’s see how we might design a Complex number class.

What we expect to be able to do is all mathematical operations between complex numbers as well as scalar multiplications and division.

- \[(1+i2)+2*(3-i5)*(i4)/(1+i3)\]
- \~(1+i2) [Conjugate]
- !(3+i4) [Modulo]
Scala allows to implicitly create constructors and attributes starting from a simple argument list associated with the class declaration.

```scala
class Complex(val re: Float, val im: Float)
```
In Java

```java
public class Complex {
    private final float re;
    private final float im;

    public Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    public float re() { return re; }
    public float im() { return im; }
}
```
Methods

› Everything in Scala is a method even operators

› Methods name can be symbols such as *, !, +, etc.

```scala
def + (c: Complex) : Complex = Complex(re + c.re, im + c.im)
```

or, taking advantage of type inference....

```scala
def + (c: Complex) = Complex(re + c.re, im + c.im)
```
public Complex add(Complex that) {
    return new Complex(this.re() + that.re(),
                       this.im() + that.im());
}
As a result...

Scala

```scala
val result = Complex(1,2) + Complex(2,3)
```

Java

```java
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex(3,4);
Complex c3 = c1.add(c2);
```

or...

```java
Complex c3 = (new Complex(1, 2).add(new Complex(3,4)));
```
Negation and Scalar Multiplication

In order to design a Complex class that is well integrated in our type system we should be able to support the following cases:

- \(-(a+ib)\)
- \(c(a+ib)\)
- \((a+ib)c\)

How can we supporting something like \(-(a+ib)\) and \(c(a+ib)\)?
Scala Unary Operators

Scala allows to define unary operators for the following method identifiers +, -, !, ~

```scala
def unary_-(()) = Complex(-re, -im)

def unary_!(()) = Math.sqrt(re*re + im*im)

def unary_~() = Complex(re, -im)
```

as a result we can write:

```scala
val result = -Complex(1,2) + ~Complex(2,3)
```
Scala Implicit Conversions

- The expression:
  
  ```scala
  val c3 = 3*Complex(5, 7)
  ```

- Is equivalent to:
  
  ```scala
  val c3 = 3.*(Complex(5, 7))
  ```

- Yet, the method to multiply a Integer to a Complex is not present in the Scala Int class

- What can we do to make the trick?
Scala Implicit Conversions

- Scala does not support Open Classes, thus allowing to add new methods to existing classes
- Yet Scala supports implicit conversions that can be used to achieve the same result
- Let's see how...
Scala Implicit Conversion

```
object Complex {
  implicit def floatToReComplex (f: Float) = new ReComplex(f)

  implicit def intToReComplex(i : Int) = new ReComplex(i)
}

class ReComplex(re: Float) {
  def * (that: Complex) = Complex(re*that.re,re*that.im)
}
```
The Result is...

\[
\text{val } c3 = 3*\text{Complex}(5, 7)
\]

is converted automatically into:

\[
\text{val } c3 = \text{ReComplex}(3).*(\text{Complex}(5, 7))
\]
class Complex(val re: Float, val im: Float) {

  // Binary Operators
  def + (c: Complex) = Complex(re + c.re, im + c.im)
  def - (c: Complex) = Complex(re - c.re, im - c.im)
  def * (f: Float) = Complex(f*re, f*im)
  def * (c: Complex) = Complex((re*c.re) - (im*c.im),
      ((re*c.im) + (im*c.re))
  )
  def / (c: Complex) = {
      val d = c.re*c.re + c.im*c.im
      Complex(((re*c.re) + (im + c.im))/d,
          ((im*c.re) - (re*c.im))/d )
  }

  // Unary Operators
  def unary_-(c: Complex) = Complex(-re, -im)
  def unary_!(c: Complex) = Math.sqrt(re*re + im*im)
  def unary_~(c: Complex) = Complex(re, -im)

  // Formatting
  override def toString() : String = {
      if (im > 0) re + "+i" + im
      else if (im < 0) re + "-i" + (-im)
      else re.toString
  }
}
Stepping Into

Scala

FUNCTIONS, CLOSURES, & CURRYING
Functions

- Scala has first-class functions

- Functions can be defined and called, but equally functions can be defined as unnamed literals and passed as values

```scala
def inc(x: Int) = x + 1
inc(5)

val vinc = (x: Int) => x+1
vinc(5)
```

Notice once again the uniform access principle
Playing with Functions

val list = List(1,2,3,4,5,6,7,8,9)
val g5 = list.filter((x: Int) => x > 5)
g5: List[Int] = List(6, 7, 8, 9)

Or with placeholder syntax

val list = List(1,2,3,4,5,6,7,8,9)
val g5 = list.filter(_ > 5)
g5: List[Int] = List(6, 7, 8, 9)
Closures

- Scala allows you to define functions which include **free variables** meaning variables whose value is not bound to the parameter list.

- Free variable are resolved at runtime considering the closure of visible variable.

- Example:

```scala
def mean(e : Array[Float]) : Float = {
    var sum = 0.0F
    e.foreach((x : Int) => sum += x)
    return sum/e.length
}
```

```scala
def mean(e : Array[Float]) : Float = {
    var sum = 0.0F
    e.foreach(sum += _)
    return sum/e.length
}
```
Currying

- Scala provides support for curried functions which are applied to multiple argument lists, instead of just one
- Currying is the mechanism Scala provides for introducing new control abstraction

```scala
def curriedSum(x: Int)(y: Int) = x + y

curriedSum(1) {
  3 + 5
}
```
Stepping Into Scala
Traits

Scala supports single inheritance from classes but can mix-in multiple traits

A trait is the unit of code reuse for Scala. It encapsulate methods and field definitions

Traits usually expect a class to implement an abstract method, which constitutes the “narrow” interface that allows to implement a rich behaviour
Ordered Complex Numbers

- Our complex numbers are not comparable

- Let’s assume that we wanted to make them comparable, and let’s supposed that we define the total order as based on the module of the complex number

- How can we implement this behavior?
Ordered Trait

- The Ordered[T] traits encapsulates the set of methods that allow to define a total ordering over a type
- All the behaviour is defined in terms of an abstract method, namely “compare”
- Classes that mix-in this trait have to implement the “compare” method

```scala
class Complex(val re: Float, val im: Float) extends Ordering[Complex] {
  def compare(x: Complex, y: Complex) = {
    if (x == y) 0
    else if (!x > !y) 1
    else -1
  }
}
```
Stepping Into Scala Type Parametrization
Type Parametrization

- Scala provides support for type parametrization and makes it available for both classes as well as traits

```scala
trait Queue[T] {
  def head: T
  def tail: Queue[T]
  def append(x: T): Queue[T]
}
```

- Scala allows to annotate the parametrized type to control the resulting type variance
If $S <: T$ is $\text{Queue}[S] <: \text{Queue}[T]$?

By default Scala makes generic types nonvariant. This behaviour can be changed using the following annotations:

- $\text{Queue}[+T]$ indicates that the the sub-typing is covariant in the parameter $T$
- $\text{Queue}[-T]$ indicates that the the sub-typing is contravariant in the parameter $T$
Who is Using Scala

Scala adoption has been propelled by some user of excellence such as:

- eDF
- Twitter
- SONY
- SIEMENS

Delivering Performance, Openness, and Freedom


STEPPING INTO DDS
How Does it Work?

- DDS is based around the concept of a **fully distributed Global Data Space (GDS)**
- Applications can autonomously and asynchronously read/written data in the GDS

Note: DDS can be seen as a relaxation of the LINDA coordination model.
How Does it Work?

- Publishers and Subscribers can join and leave the GDS at any time
How Does it Work?

- Publishers and Subscribers express their intent to produce/consume specific type of data, e.g., Topics
Subscriptions are **matched** by taking into account topics (name, data type and QoS)
How Does it Work?

- Subscriptions are dynamically matched and Data flows from Publisher to Subscribers

Tweeting with OpenSplice DDS

Starting to Tweet with OpenSplice DDS
A “Tweet” with DDS

**Topic:**
- Unit of information exchanged between Publisher and Subscribers.
- An association between a unique name, a type and a QoS setting

Tweet

- Name
- Type
- Topic
- QoS

TweetType

{ Reliable, Persistent, ... }
A “Tweet” with DDS

**Topic Type:**
- Type describing the data associated with one or more Topics
- A Topic type can have a key represented by an arbitrary number of attributes
- Expressed in IDL (or XML)

```cpp
struct TweetType {
    string name;
    string tweet;
};
#pragma keylist Tweet userId
```
Tweeting With DDS

- Tweeters come...
- Tweeters go...

Global Data Space

@bird
@oiseau
@lind
@ocell
@lintu
@vogel
Tweeting With DDS

- Tweeters follow (subscribe) to other tweeters

- DDS Global Data Space

- Tweeters: 
  - @bird
  - @oiseau
  - @vogel
  - @lintu
  - @ocell

- Followed by: 
  - {@oiseau, @bird, ocell}
  - {@ocell, @oiseau}
  - {@vogel}
Tweeting With DDS

- Each Tweeter can be represented with a unique topic

- @bird
- @oiseau
- @lind
- @vogel

- @birdTopic
d- @oiseauTopic
d- @lindTopic
d- @vogelTopic

- @ocell
- @lintu

- `{@oiseau, @bird, ocell}`

Tweeting With DDS

- ...or each Tweeter can be represented with a unique partition
- All tweeters use the same topic

Note: DDS partitions can be specified by using wildcards, e.g. tweeter.*
...or all Tweeters use the same topic

Content filtering or queries are used to extract relevant data

Tweeting With DDS
Delivering Performance, Openness, and Freedom

OpenSplice DDS


WHO IS USING DDS
Some Use Cases

Defense & Aerospace

- Naval Combat Management Systems
- Submarines
- Vetronics
- Tactical Links
- Simulation
- Cybercrime
- Flycatcher Systems
- Data Fusion
- Battle Transformation Center

Some Use Cases

Transportation
- Drones
- Air Traffic Control & Management
- Metropolitan Transportation

Financial Services
- Automated Trading Firms
- Risk Management Firms
The Escalier Project

- A new OpenSplice DDS technology incubator recently launched to develop a Scala binding for OpenSplice DDS
- The Scala Binding is being developed by enhancing the current Java Binding
- Project hosted on google code first public release by the end of June

http://code.google.com/p/escalier/
object TweetWriter {

    def main(args: Array[String]): Unit = {
        Runtime.start()

        // Create a Domain Participant in the TweetDomain
        val tweetDomain = DomainParticipant("Tweet")

        // Create a Tweet Topic (in the TweetDomain) with default QoS

        // Create a Publisher (in the TweetDomain) with default QoS
        val pub = Publisher("TweetPublisher", tweetDomain)

        // Create a DataWriter for writing Tweet
        val writer = DataWriter[Tweet]("tweetWriter", t, pub)

        var tweet: Tweet = new Tweet
        tweet.name = args(0)
        val count = args(2).toInt

        for (i <- 1 to count) {
            tweet.msg = args(1) + " #" + i
            println(tweet.name + ":" + tweet.msg)
            writer.write(tweet)
            Thread.sleep(1000)
        }
    }
}
Twitting

object TweetWriter {

    def main(args: Array[String]): Unit = {
        Runtime.start()

        // Create a Domain Participant in the TweetDomain
        val tweetDomain = DomainParticipant("Tweet")

        // Create a Tweet Topic (in the TweetDomain) with default QoS

        // Create a Publisher (in the TweetDomain) with default QoS
        val pub = Publisher("TweetPublisher", tweetDomain)

        // Create a DataWriter for writing Tweet
        val writer = DataWriter[Tweet]("tweetWriter", t, pub)

        var tweet: Tweet = new Tweet
tweet.name = args(0)
val count = args(2).toInt

    for (i <- 1 to count) {
        tweet.msg = args(1) + " #" + i
        println(tweet.name + ":> " + tweet.msg)

        writer.write(tweet)
        Thread.sleep(1000)
    }

}
Pick your “write”

- We can also write as:

```python
writer.write(tweet)  =>  writer write tweet
```
Pick your “write”

- Or if we want to let treat a writer like an actor:

  writer ! tweet
Pick your “write”

And if we have a lot to tweet:

writer ! (tweet1, tweet2, ..., tweetn)
Twits Reader

object TweetsReader {
  
def main(args : Array[String]) : Unit = {
    // Start the Runtime
    Runtime start()
    // Create a Domain Participant in the TweetDomain
    val tweetDomain = DomainParticipant("TweetDomain")
    // Create a Tweet Topic (in the TweetDomain) with default Qos
    // Create a Subscriber (in the TweetDomain) with default Qos
    val sub = Subscriber("TweetSubscriber", tweetDomain)
    // Create a DataReader for reading Tweets
    val reader =DataReader[Tweet]("tweetDataReader", t, sub)

    var tweet: Tweet = new Tweet
    tweet.name = @acorsaro
    tweet.msg = "Stepping into Scala!"

    while(true) {
      val data = reader.read()
      // Print received tweets
      data.foreach((d: Tweet) => println(d.name + ": " + d.msg))
      // Sleep 1 sec
      Thread.sleep(1000)
    }
  }

  
}

Pick your “read”

- We can also write as:

```scala
val data = reader.read() => val data = reader read
```
Pick your “read”

- Or we can be a bit more functional...:

```java
reader.forEach((t: Tweet) => println(t.name + " : " + t.msg))
```
Setting QoS

Example

- Setting Durability and History on the Topic Qos

```scala
val qos = TopicQos() <= PersistentDurability() <= KeepLastHistory(tweetHistory)
val topic = Topic[Tweet]("Tweet", qos, pub)
```
Stepping Into Demo Time
OpenSplice DDS

The Universal Data Bus

Summing Up

- A good type system can help in the development of both traditional and distributed applications, as it improves code quality, productivity, safety and runtime efficiency.

- Scala and OpenSplice DDS are a very powerful combination since provide scalable language and middleware mechanism for building large scale complex distributed systems.