DISTAL: Domain-specific Language for Implementing Distributed Algorithms

Master Thesis Report

Pamela Delgado

Distributed Systems Laboratory LSR
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne

Supervisors: Zarko Milosevic, Martin Biely
Professor: Prof. André Schiper
External Expert: Prof. Pascal Felber

Lausanne, June 2012

Abstract

Distributed algorithms research focuses on interactions and communication between independent systems and processors in diverse scenarios. These algorithms are usually written as abstract pseudo-code, and turning them into compilable and runnable code is a complex and error prone task. This is in part due to the lack of expressiveness for representing the distributed algorithms abstractions in currently available tools and libraries. We present DISTAL, a domain-specific language for implementing distributed algorithms, as a library on top of Scala that allows the user to express and fully implement distributed algorithms in a high level, pseudo-code-like manner. Validated with a Paxos and its MultiPaxos variant along with a batching optimization, this library exhibits the capability of allowing a quickly implementation of these algorithms, while retaining their expected behavior and properties.

Keywords domain specific languages, distributed algorithms, Scala

Contents

List of Figures

List of Algorithms

List of DSL code examples

1 Introduction

Distributed systems are nowadays at the core of increasingly diverse applications, from highly scalable parallel systems to mobile data processing units. As a result, a number of non trivial challenges is posed to the computer science research community. On one hand, new technologies focus on efficient, reliable and scalable communication in complex networks of processing nodes, and on the other hand researchers study and propose distributed algorithms that are capable to address these problems. Examples of this type of algorithms are, reliable message passing, agreement algorithms in presence of byzantine failures, consensus algorithms, etc.

While these algorithms focus on essential abstract properties such as correctness, validity or termination, they are written in an implementation-agnostic way. Therefore, these algorithms are not biased by technological specificities, and are also simpler to understand as they only reflect a handful of key ideas. Such algorithms are typically expressed in a high level pseudo-code, which allows them to be simple and expressive, fitting into a few lines of code [4]. However, turning these algorithms into compilable code is a highly complex, time-consuming and error-prone task. Furthermore, the way an algorithm is implemented and the tools a given implementation may use can affect the performance, and some important assumptions non covered in theory, might raise problems in practice.

The lack of adequate libraries or frameworks for facilitating the tasks of developing, prototyping and deploying distributed algorithms forces implementors to devote a considerable amount of time and effort to turn an algorithm into real code. Even in high-level programming languages, there is a mismatch between the required expressiveness of an algorithm, and the available abstractions provided by a given language.

In this project we propose a domain specific language (DSL) to express distributed algorithms, implemented as a library in Scala, and that can be executed on top of the Akka[1] framework. The goal of this DSL is to help programmers bridging the gap between abstract algorithms and actual code, thus enabling a simple transition to a fully developed and runnable implementation. This project focuses specifically in the following aspects of distributed systems:

Group communication algorithms Distributed systems is a wide area and in this work we want to focus on distributed algorithms of the group communication type, namely: Consensus, Paxos, failure detector components, reliable broadcast, total order broadcast, atomic broadcast, among others [4, 11].

Application layer Using an existing component for the network layer and message passing, the project lies in the application layer, focusing more on the developer side in terms of expressiveness and facility to verify, deploy and implement algorithms in a high level fashion.

[1]http://akka.io/

The main contributions of this work can be summarized as follows:

i) We conduct an analysis of different ways of expressing distributed algorithms and their commonalities.

ii) We provide the design of a Domain Specific Language for distributed algorithms, using Scala as host language, whose main features include:

   – An abstraction for events and messages
   – The ability to mix with real Scala code, being an internal DSL
   – A framework structure that helps the user in bootstrapping and deploying (set up of peers and starting point, system generator, configuration files)

iii) We validated and experimented our approach with common use cases, including Paxos to MultiPaxos implementation, an optimization with batching, and a test comparison with JPaxos [15].

The remainder of the report is organized as follows: In Section 2 we introduce the fundamental concepts about DSLs and the analysis of commonalities found in mainstream distributed algorithms. In Section 3 we introduce the design of our DSL for distributed algorithms, and in Section 4 we describe its implementation. The experimentation and evaluation of our approach is detailed in Section 5. Related work is presented in Section 6 before concluding in Section 7.

2 Background

Before describing our proposal we present in this section some fundamental concepts related to Domain-specific languages and an analysis of distributed algorithms that helped modeling the DSL features presented in next section.

2.1 Domain-specific languages

As we are in the information era, nowadays all types of domains start searching for better ways of using the technology to their favor. Starting with areas outside computer science like medicine, biology, architecture, astronomy, etc, not to mention several industry applications, we find a wide world of non-programmers who want to tell the computer how to compute results, in other words, programs. People started using more and more domain-specific languages to achieve their goals without having to learn a general purpose programming language [26].

The same phenomenon happens also inside computer science, where standard programming languages are sometimes not enough. The growing number of tools that helps scientists and developers to perform software verification, implementation of parallel programs (Liszt [7], OptiML [24], Green Marl [12]), database management systems (Pulse [1]) and probabilistic programming, are just an example of the need for DSLs.

One can think of a DSL as being a small programming language focused to a specific area, it usually provides a way of representing abstractions, unlike a general purpose language. According to the way it is implemented, a DSL can belong to one of two types of DSLs: external and internal [8]. While the first kind, also called *standalone* is not necessarily dependent of a general programming language, its implementation includes also a parser and a compiler or interpreter. The second type of DSLs, internal, is embedded in a host language and consequently, its syntax is somehow limited.

As the DSL for distributed algorithms defined by its requirements is an internal DSL, we present a rough analysis and comparison of the most popular JVM host languages for developing internal DSLs based on [10]. JRuby, Groovy and Clojure are alike in the sense that all of them provide meta programming, macros and dynamic typing, while the syntax flexibility in Ruby and Groovy are higher than Clojure. Scala on its down side, does not support dynamic typing nor provides macros for meta programming, but there are a series of advantages of using it as a host language, to name the most relevant: implicit conversion, pattern matching, case classes, default arguments, flexible syntax with dot and parenthesis, static types, host for object oriented and functional programming. Embedded DSls in Scala have also been coupled with code generation in [21].

2.2 Distributed algorithms analysis

We studied the style common patterns and conventions for writing distributed algorithms in the literature, and we came to the conclusion that there is not a single or universally agreed way of expressing them. While this fact poses challenges with respect to the expressiveness of the code that implements these algorithms, at the same time it provides certain freedom for defining a DSL syntax.

We analyzed a set of algorithms and the way they are expressed, among them [2], [4], [14], [20], [3], [22], [6], [11], and we evidenced that despite the variety of syntax in their pseudo-code, they share elements in common. Generally speaking, distributed algorithms are expressed as processes reacting to events, message types and their contents and quantity. The lack of expressiveness of languages, frameworks and platforms for dealing with these concepts, can be identified as one of the main obstacles when trying to implement distributed algorithms. The mismatch between abstractions such as events or messages in pseudo-code, and the elements available in programming languages, is the first issue that needs to be tackled. From that point on, algorithms require the ability of specifying how messages are exchanged, when an event is triggered, and what actions should be executed on those circumstances.

This is precisely the starting point for the definition and modeling of the DSL, and in the following two sections we describe the solution including a detailed description of the DSL features and its implementation.

3 DISTAL Language Design

We designed a DSL for developing distributed algorithms, with a special focus on group communication. The functionalities of the DSL as a language, were derived from a set of key directives, as a result of our analysis of existing algorithms in the literature.

Firstly, the DSL should define a high-level language and be as expressive as possible, allowing to write algorithms in a form very close to pseudo-code. As we have seen in the previous section, algorithms in the literature can be very different in form, but they share many notions that can be abstracted as common elements. By defining a language that is very close to pseudo-code, we minimize the cost of translation in terms of time and difficulty.

Secondly, it is required to allow re-usability whenever it is possible. We identify three levels of re-usability:

Algorithmic level When studying common distributed algorithms we found that they are typically expressed in layers, for example an *Atomic Broadcast* algorithm could use a *Reliable Broadcast* algorithm underneath. Consequently the DSL should allow defining algorithms in such a way that upper-layer algorithms may use lower-level ones.

Code level This DSL library should allow the developer to seamlessly mix DSL features with the host language code and libraries. In the case of our DSL implementation, the host language is Scala, is very expressive and provides collection manipulation features that allow the developer, for example to iterate over a list of messages in a couple of code lines. Reimplementing these functions as a feature as well as other language elements such as loops not only would not provide any advantage to the DSL but also it would make it more complex to develop and, if it is not carefully implemented, less efficient. Therefore we promote code re-usability within the host language as an important DSL feature.

Parser/interpreter level As mentioned before in Section 2 there are two types of DLSs: external and internal. Although external DSLs allow full freedom for the definition of the language syntax, they may require defining a parser, parser combinator, compiler and even interpreter. Furthermore, to accomplish the previous goal of code re-usability, it is far simpler and convenient to implement an internal DSL, for which a parser combinator for instance is discarded. Although some internal DSLs define a small yet complete language they usually include an interpreter for it. In the case of our DSL, implementing compilers, parsers and interpreters is avoided, we completely reuse those of the host language.

Following an analysis of common ways of expressing distributed algorithms, we defined the key features of our proposed DSL. At the very core of the DSL we define *Events* and *Messages*, described in detail in the following two subsections. While in these latter we show pieces of DSL code, a complete Reliable

Broadcast code example can be found in the last subsection and a full Paxos implementation is also presented in evaluation.

3.1 Events

Events are the main component of this DSL, this section presents features and cases related to events and how to use them. Distributed algorithms are typically expressed in terms of events being triggered, and actions launched when an event is detected.

Events: An event is an occurrence triggered by an algorithm at some point of its execution. In the wake of an Event, some action is typically executed by the algorithm, receiving a message for processing. Algorithms in the DSL are able to listen to event occurrences with the `UPON EVENT` keywords. In the next example, the algorithm will execute the action inside the `DO` clause, every time the `MyEvent` event is triggered. This action is defined as an anonymous function that receives a message and can process it inside: `(m: MyMsg) =>{<do something with m>}`.

```
UPON EVENT MyEvent() DO (
    (m: MyMsg) =>{...})
```

DSL code 1: Events in DISTAL

Conditions: Sometimes users want to verify if a condition holds before the action is executed. We specify a condition using the `WITH` keyword followed by a condition function. In the following example the condition is that the message attribute `k` should be larger than some internal value `myk` (this value could represent the number of a view instantiation, for example).

```
def biggerK(m: MyMsg) = (m.k > myk)
UPON EVENT MyEvent() WITH biggerK DO ((m: MyMsg)=>{})
```

DSL code 2: Conditions in DISTAL

Counting: Another variant of conditional consists in controlling a condition that holds a given number of times. Then, when some threshold is met, the algorithm proceeds to execute the action. This is specified using the `TIMES` keyword after the `WITH` condition. This is a useful feature particularly for algorithms that require verifying conditions on a number of messages received by peers, for instance half of them in the next example.

```
UPON EVENT MyEvent() WITH condition TIMES n/2 DO ...
```

DSL code 3: Counting in DISTAL

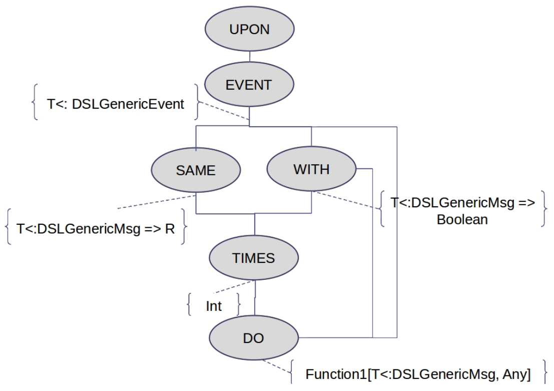

Figure 1: Event definition options

Comparing messages: Sometimes the content of the messages matters, and we want to execute an action after we received n messages of the same kind. By same kind we refer to same value for an attribute (or a combination of them), and this can be specified using the `SAME` keyword followed by the function that provides the message value to be compared, and the number of times it is expected to appear before executing the action (using the `TIMES` keyword). In this example we filter messages with the same vote value. When the threshold `n` is met for a list of messages of the same kind, this list is sent as parameter to the event action.

```
def voteValue(m: VoteMsg) = m.vote
UPON EVENT Vote() SAME voteValue TIMES n DO(
  (msgs: List[VoteMsg]) =>{...})
```

DSL code 4: Comparing messages in DISTAL

Figure 1 summarizes different ways of building an event including the type of parameters expected in each stage of the DSL.

3.2 Messages

The second main component of the DSL, messages, is closely related to events. In this subsection we explore how to define as well as how to use them to activate and process Events.

Message Definition To be able to define the events as shown previously, the user would need to create classes defining the messages, with their respective attributes, and the events. The only condition is to extend DSL Message and Event classes respectively. There is also a definition for an empty message, and if desired, attributes can contain default values.

```
case class MyMessage(att1: Int=0, att2...) extends DSLMessage
case class MyEvent(msg: MyMessage) extends DSLEvent(msg)
```

DSL code 5: Message definitions in DISTAL

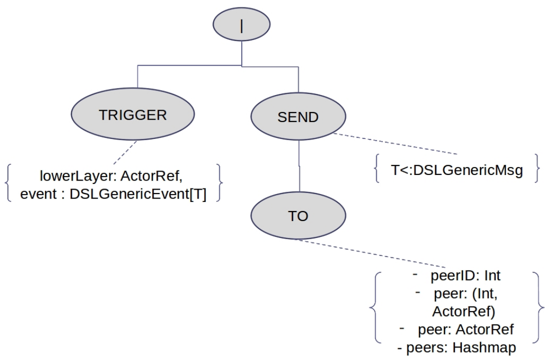

Figure 2: Execution line options

Another important abstraction present in the DSL is activating events by sending messages. We provide two ways, send and trigger.

Send To send messages that activate an event across the network, we typically use `SEND`, this is translated by the DSL to Akka message-passing.

```
def myMsg = MyMessage(0,ack,...)
SEND MyEvent(myMsg) TO peerid
```

DSL code 6: Sending messages in DISTAL

Trigger The other option is to `TRIGGER` an event internally, for instance from a lower layer algorithm, for example from a reliable broadcast instance.

```
TRIGGER(rebInstance,MyEvent(myMsg))
```

DSL code 7: Triggering events in DISTAL

In both `SEND` and `TRIGGER` is possible to specify the destination with the peer ID, an actor reference and a list of peers.

Figure 2 summarizes the different ways of building an executable line of the DSL including all the options for specifying a peer in `SEND`.

3.3 From pseudo-code to DSL

Now that we have explained the DSL features we can visualize an implementation of a simple and concrete distributed algorithm as an example. Consider reliable broadcast algorithm, slightly modified from [11], with two events defined in algorithm 1.

If we were to port this to DISTAL we have the following implementation, we also include the definition of the messages which is trivial [2].

[2]DSLActor provides also `myPeers` and `parent` variables

Algorithm 1 Reliable broadcast

```
1: Upon Event SendReliableBroadcast($m$)
2:     SEND $m$ TO my peers
3:
4: Upon Event ReliableBroadcast($m$)
5:     if $m$ not in $delivered$
6:     SEND $m$ TO my peers
7:     $delivered := delivered + m$
8:     DeliverRBroadcast($m$)
```

```
// Messages
case class BroadcastMsg (id: Int, content: String) extends DSLGenericMsg
// Events
case class SendReliableBroadcast(bmsg: BroadcastMsg = null) extends DSLGenericEvent(bmsg)
case class ReliableBroadcast(rbmsg: BroadcastMsg = null) extends DSLGenericEvent(rbmsg)
case class DeliverRBroadcast(dmsg: BroadcastMsg = null) extends DSLGenericEvent(dmsg)
// Main algorithm actor class
class ReliableBroadcast extends DSLProtocol{
  val delivered: List[BroadcastMsg] = List()
  UPON EVENT SendReliableBroadcast() DO (
      (m: BroadcastMsg)=>{
    | SEND ReliableBroadcast(m) TO myPeers
  })

  UPON EVENT ReliableBroadcast() DO (
    (m: BroadcastMsg)=>{
        if(!delivered.contains(m)){
        | SEND ReliableBroadcast(m) TO myPeers}
        delivered = delivered ++ List(m)
        | TRIGGER(parent,DeliverRBroadcast(bmsg))
        }
    }
  )}
```

DSL code 8: Reliable broadcast implementation

Observing and comparing the algorithm pseudo-code and the implementation we can note three important properties of the DSL. 1) It does not grow in number of lines. 2) At the same it allows to mix DSL and Scala code inside actions, for instance the addition of a new message to the list of delivered messages. 3) Moreover, the implementation is expressive enough to understand the purpose of the algorithm at a glance.

4 Implementation

While the DSL design and features were presented in the previous section, this section will cover the implementation of the DSL itself, some particularities and the architecture of DISTAL as a library.

4.1 How it works

As we explained before, Scala provides internal DSLs with some built-in features that enable them to use syntactic sugaring. These features are mainly implicits, optional dots, high order functions and case classes, that let us build from the user code an abstract tree composed of classes, objects and methods. This tree is built as the DSL code is written.

```
UPON EVENT MyEvent() WITH condition DO (
  (m: MyMsg) =>{...
    | SEND m TO peer
})
```

DSL code 9: Conditional event example

The two main classes of the implementation of the DSL are the `EventBuilder` and the `LineBuilder`. If we take the previous example of an event definition starting with `UPON`, which is an object, by means of implicit we create an `EventBuilder`, and the only way of doing this using the object `EVENT`. Taking the event and its corresponding type, the latter will create an `UponBranch` that can contain objects of types `WITH`, `SAME` or `DO`. These branches can contain subbranches on their own, to define objects like `TIMES` or `DO`. Finishing the construction with an action defined in terms of a function applied to a message, this abstract tree is created in order to define an event settings. Internally the `DSLActor` will create an `EventDefinition`, saving all these values to process them when needed, and adding it to a Hashmap of events. The message passing through the network is taken care underneath by Akka.

Figure 3 shows the tree built from the example shown previously.

Similarly, for a DSL execution line we have the `LineBuilder`, represented by a special object `|` to distinguish a DSL line from a normal Scala execution. This object is in charge of the definition of `SEND`, with its own branch and options, and `TRIGGER`. Since these objects typically lie in an event action, their corresponding apply methods in the DSL implementation are translated to executable code, in the case of the example above it will send a message to a peer. By its nature, a DSL execution line can be also part of a normal Scala method or even just lie inside a class.

4.2 DSL particularities

As we stressed before, internal DSLs are limited by the host language. In the case of Scala and this Distributed Algorithms DSL, there were some challenges

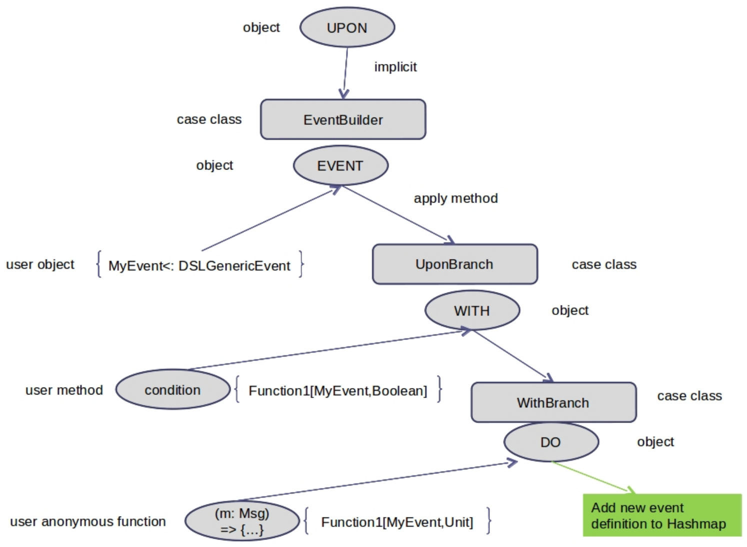

Figure 3: Abstract tree built from DSL code 9 example

and particularities to consider during the implementation.

The fact that in Scala there are no means to use macros [10] (to dynamically define new types for example) like in other languages such as Ruby, made the DSL implementation limited, from a syntax point of view. For instance in the definition of actions inside events, a function that receives an argument of the same type as the event message is needed to be declared explicitly like an anonymous function. When developing the DSL several options where taken into account before choosing the more expressive among the feasible solutions.

Another example of this syntax limitation is the `|` commodity character used to build an executable line. Since an object for the implicit was needed in any case, we have chosen this character because is less intrusive for the user code.

A challenging part of the implementation was to deal with the statically typed characteristic of Scala. As we saw before, actions inside events are defined in terms the type of message that it receives, meaning new types defined by the user. These types presented a difficulty when creating the Event in the DSL, because we did not know those types in advance to tell the compiler. This is why all objects in DISTAL implementation are type parametrized. Furthermore, the new types created by the user both messages and events have to be subtypes from `DSLGenericMsg` and `DSLGenericEvent` (known types to the DSL).

A special case with static types happened for the anonymous function: since the first type parameter is covariant we could not take as a parameter a super type, like `DSLGenericMsg`. A combination of *Nothing* and a cast when applying this function to the incoming message, together with type parametrized classes and objects was the trick that solved this issue.

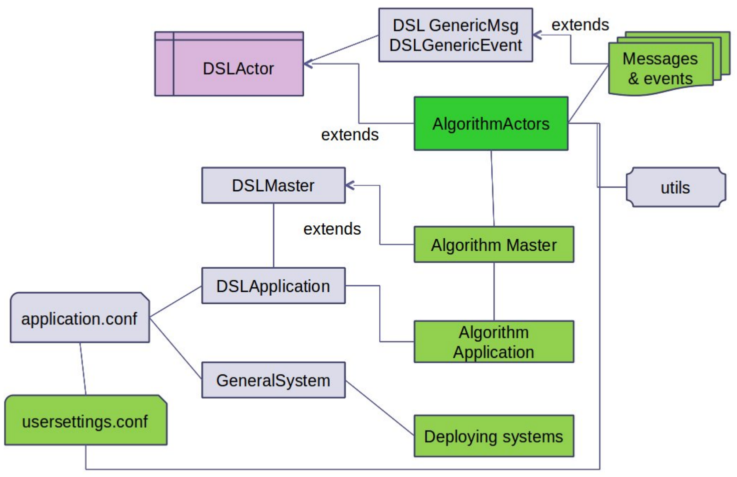

Figure 4: Architecture of DISTAL library

An inherent drawback, from the user point of view, of internal DSLs is that the exceptions he/she might get are not always meaningful. In our DSL we tried to design in the simplest yet complete syntax to avoid confusion.

4.3 Architecture

Although the core of the DSL library explained in the previous two subsections lies inside the `DSLActor` class, there is a set of classes outside that also form part of the library, and are devoted to facilitate the deployment task to the user. The architecture of the DSL library and how user classes are related to it are shown in Figure 4.

In this figure the green elements correspond to user defined classes and objects, and the blue ones to the DSL library elements. The main algorithm will be implemented in a class, corresponding to `AlgorithmActors` in the figure, along with its events and messages definition. These classes should extend `DSLActor`, `DSLGenericEvent` and `DSLGenericMsg` correspondingly.

For the implementation to be able to run we need to create Akka systems in each node, actors (or instances) of the main algorithm and define a configuration file with all the parameters needed for Akka. The `DSLApplication` is the starting point for the run, it is type parametrized with the class type of actors and the Application Master and it will be in charge of creating the actors using the configuration file and to call the master bootstrapping. The user should define an Algorithm Application, bootable or with a main class, that creates a new `DSLApplication` and pass an Algorithm Master object.

Another important block for the deployment is `DSLMaster`, which is in charge of all the bootstrapping, meaning the exchange of peers information, waiting for acknowledge, and the start and stop points of the algorithm. The corresponding user Algorithm Master should extend this class and if desired override the `startAlgorithm` method.

To be able to create Akka actors, an `ActorSystem` is needed in each machine node, this is wrapped in `GeneralSystem` class, for which the user will just need to create a new class giving as parameters the name of the system, its configuration setting name and the configuration file path.

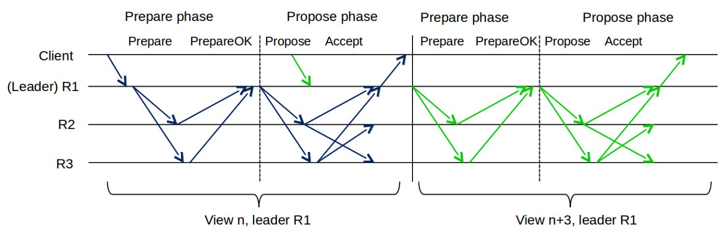

Figure 5: Paxos execution scenario

5 Evaluation

This section presents the evaluation of DISTAL including the implemented algorithm, the settings for testing, and the results for both expressiveness and performance points of view.

5.1 Paxos

Among the different distributed algorithms present in the literature, we have chosen Paxos [16, 17] to implement and evaluate the DSL. There are several reasons for this choice, first of all it is a complex and well known algorithm key to consensus, secondly because it has several variants with which we could test how the DSL is helpful when comparing different alternatives of the same problem, and finally because we count with almost the same settings and the source code of a Java implementation of it: JPaxos [22].

The main idea of Paxos is to solve agreement and perform consensus over a value, usually proposed by clients between a given number of replicas, in an asynchronous non byzantine environment. It consists of two phases, prepare and propose. During prepare phase the leader[3] of a given *view* sends a *Prepare* message and when it gets a majority of *PrepareOK* responses from the other replicas, it sends one of the values proposed by the clients within a *Propose* message. When a replica receives a proposal it answers with an *Accept* to all the peers, and finally each replica can decide on a value once it has received *Accept* from a majority of its peers. Figure 5 illustrates the execution of two consequent views in which the first replica is the leader.

In the Algorithm 2 and DSL code 12 we present the full pseudo-code and DSL code respectively of a detailed implementation of the algorithm described above used in the evaluation. To increase clarity, failure detector code has been omitted.

[3]several leader election algorithms are also present in literature

Algorithm 2 Paxos algorithm

```
Initialization:
    $view \leftarrow 0$   // used to recognize voting rounds
    $(lastview, lastvalue) \leftarrow \{0, Nil\}$   // last accepted value and its view
    $procId \leftarrow ID$   // ID of the process
    $accepted \leftarrow \{Nil\}$   // set of processes that accepted the value in current view
    $valuesProposed \leftarrow Nil$

PreparePhase:
    if valuesProposed has at least one message and prepare or propose phase
        $view \leftarrow nextMyView$
        send Prepare<$viewm$> where $viewm \leftarrow view$ to all

ProposePhase:
    $lastvalue \leftarrow valuesProposed$
    $lastview \leftarrow view$
    send Propose<$viewm$,$valuem$> where $viewm \leftarrow lastview$; $valuem \leftarrow lastvalue$ to all

Upon Prepare<$viewm$> where $viewm \geq view$ from $p$
    if $viewm > view$ then
      advanceView($viewm$)
    $view \leftarrow viewm$
    send PrepareOK<$viewm$,$\{viewp, valuep\}$> where $viewm \leftarrow view$; $\{viewp, valuep\} \leftarrow \{lastview, lastvalue\}$ to $p$
    if process is leader of view
        leave Propose or Prepare phase

Upon PrepareOK<$viewm$,$\{viewp, valuep\}$> with $viewm = view$ times majority
    $\{viewv, value\} \leftarrow \{viewp, valuep\}$ from PrepareOK with highest $viewp$
    begin Propose phase

Upon Propose<$viewp, valuep$> where $viewp \geq view$ from $p$
    $accepted \leftarrow nil$
    if $viewp > view$ then
      advanceView($viewp$)
    else if $viewp = view$
      $\{viewv, value\} \leftarrow \{viewp, valuep\}$
    send Accept<$viewm, valuem$> where $viewm \leftarrow viewp$; $valuem \leftarrow valuep$ to all

Upon no decision taken and the leader crashed
    begin Prepare phase
```

Algorithm 3 Paxos algorithm - continuation

```
Upon Accept<$viewp, valuep$> where $viewp \geq view$ from $p$
  if $viewp > view$ then
    advanceView(viewp)
  else if $viewp = view$
    if $viewv! = viewp$ then
      execute Upon Propose<$viewp, valuep$>
    $accepted \leftarrow accepted \cup \{p\}$
    if $accepted$ contains majority of processes then
        decided on value
        leave propose or prepare phase
        if procID is leader of view
          preparePhase

Upon ClientRequest <$value$> from c
  $clientProposalValue \leftarrow value$
  if leader(view) = procID
      send RedirectLeader<$leader$> where $leader = leader(view)$ to c
  else
      add value to valuesProposed
      if (phase.equals("NONE"))
          begin prepare phase
```

```
class Paxos extends DSLActor {
 // Initialization
 var view = 0
 var lastview: Int = 0
 var lastvalue: Value = null
 var accepted:  List[(ActorRef,Int)] = Nil
 var phase: String = "NONE"
 var valuesProposed: Queue[Value] = new Queue()
 def preparePhase = {
     if(valuesProposed != null && valuesProposed.size > 0 && phase.equals("NONE")){
      phase = "PREPARE"
      view = nextMyView
      | SEND Prepare(PrepareMsg(view)) TO myPeers
     }}

  def proposePhase = {
    phase = "PROPOSE"
    lastvalue = valuesProposed.dequeue()
    lastview = view
    | SEND Propose(ViewValueMsg(lastview,lastvalue)) TO myPeers
  }

  def biggerEqualView(m: PrepareMsg) = m.viewm >= view;
  UPON EVENT Prepare() WITH biggerEqualView DO (
  (m: PrepareMsg)=>{
    if (m.viewm > view)
      advanceView(m.viewm)
      view = m.viewm
      | SEND PrepareOK(PrepareOKMsg(view, (lastview,lastvalue))) TO sender
      if(leader(view)!= myID)
        phase = "NONE"
  })

  def sameView(m: PrepareOKMsg) = (m.viewm == view && phase.equals("PREPARE"))
  UPON EVENT PrepareOK() WITH sameView TIMES majority DO (
      (msgs: List[PrepareOKMsg])=>{
        val lastviewvalue = (msgs foldLeft msgs.head.viewvaluep) {
          (x, y) => if(x._1 < y.viewvaluep._1) y.viewvaluep else x }
        lastview = lastviewvalue._1
        lastvalue = lastviewvalue._2
        proposePhase
  })

 def biggerEqualView(m: ViewValueMsg) = (m.viewm >= view)
  UPON EVENT Propose() WITH biggerEqualView DO (
   (m: ViewValueMsg)=>{
   accepted = Nil
   if (m.viewm > view)
    advanceView(m.viewm)
   else {
     lastview = m.viewm
     lastvalue = m.valuem
     | SEND Accept(ViewValueMsg(m.viewm,m.valuem)) TO myPeers
  }})

  UPON EVENT Accept() WITH biggerEqualView DO(
   (m: ViewValueMsg)  => {
   if (m.viewm > view)
    advanceView(m.viewm)
   else {
     if(lastview != m.viewm)
       | TRIGGER(self,Propose(ViewValueMsg(m.viewm,m.valuem)))
     accepted = accepted ++ List((sender,myID))
     if(accepted.size == majority){
     | SEND Decided(lastvalue) TO server
     phase = "NONE"
     if(leader(view) == myID)  preparePhase
     }}})

  UPON EVENT ClientRequest() DO (
    (v: Value)=>{
     if(leader(view) != myID){
       | SEND RedirectLeader(NewLeaderMsg(myPeers(leader(view)))) TO sender
     } else {
      valuesProposed.enqueue(Value(v.clientID, v.sequenceID, v.value, sender))
      if (phase.equals("NONE"))
        preparePhase
     }})

  UPON EVENT LeaderCrashed() DO (
    (_)=>{ preparePhase })
```

DSL code 10: Paxos implementation

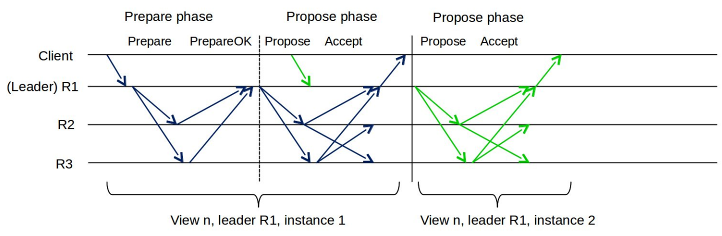

Figure 6: MultiPaxos execution scenario

5.1.1 MultiPaxos

In order to test the facility to migrate from this simple implementation of Paxos we studied one of its variants: MultiPaxos [17, 5]. Since in most of Paxos implementations the leader tends to be stable (in terms of crashing) and given that the performance in Paxos (and most of distributed algorithms) is driven by the quantity of messages sent across the network, a variant called MultiPaxos emerges as an option to solve this problem. In MultiPaxos, once the prepare phase is finished, the leader would send propose, as normal Paxos. However, for proposing the next client value it would not need to install a new view (prepare phase) and it would send directly *Propose*. This behavior is illustrated in figure 6.

The following listing shows the set of changes needed to transform our Paxos implementation from above to a MultiPaxos variant. The most important change is 4), adding a call to `proposePhase` in `Accept` event. Changes to messages to include the instance number are not included.

```
  //1) Add instance number to differentiate between proposal phases, send instance in messages
  //2) Add a PREPROPOSE phase
  var instance = 0
  def proposePhase = {
    if(phase.equals("PREPROPOSE") && valuesProposed != null && valuesProposed.size > 0){
      instance = instance + 1
     | SEND Propose(ViewValueMsg(view,instance,... }
...
  // In PrepareOK - last lines
    phase = "PREPROPOSE"
    proposePhase })
...
  //3) Update Accepted variable to a Hashmap with key being the instance number
  var accepted:  HashMap[Int,List[String]] = HashMap()
...
  // In Accept when adding a new peer
    var newList = List(sender.path.name)
    if(accepted.contains(m.instance))
      newList = newList ++ accepted(m.instance)
    accepted.put(m.instance, newList)
    if(newList.size == majority){ ...
  //4) After deciding on a value, the leader should start again Propose phase
```

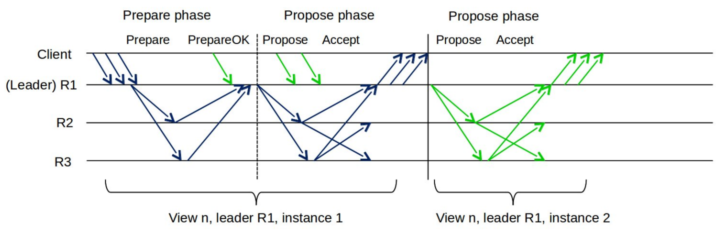

Figure 7: BatchiPaxos execution scenario

```
// In Accept last lines
  if(leader(view) == myID) {
    phase = "PREPROPOSE"
    proposePhase
  } else phase = "NONE"
```

DSL code 11: Modifications to Paxos to get MultiPaxos

Adding these changes to the code was straightforward, it barely increased (less than 10) the number of lines and the expressiveness was still maintained.

5.1.2 BatchiPaxos

Two optimizations to MultiPaxos studied in [15] are *Batching* and *Pipelining*, concluding in their results that the first one affects performance. This is why we chose to implement also this optimization and run tests on it.

As we mentioned before, the bulk of performance lies in the network, therefore the number of messages sent is key for latency and throughput, and Batching takes advantage of this by appending together several client requests and performing just one *Propose* phase for a given message size threshold. As shown in Figure 7, client requests may come at different stages and the leader would keep a queue to send them in batches.

The following listing shows the set of changes needed to add Batching optimization to the previous MultiPaxos implementation. Changed Messages are accordingly to receive a list of values each and this is not included in the listing to maintain clarity.

```
//1) Add a Batchsize variable, obtain its value from configuration file
//2) Update LastValue variable to a List of client values
var lastbatchvalues: List[Value] = Nil
var batchsize: Int = _
var msgBatch: Int = 1
override def initialization ={
  batchsize = config.getInt("batchsize")}
//3) The first time a client request arrives, calculate the number of requests needed to fill the batch size
def preparePhase = {
  if(valuesProposed != null && valuesProposed.size > 0){
      msgBatch = batchsize/valuesProposed(0).value.length
...}
  //4) Add a control to Propose only if the threshold is met and in the message include the n first requests.
  // In proposePhase
    if(valuesProposed.length >= msgBatch){
      phase = "PROPOSE"
      lastbatchvalues = Nil
      for(i <- 0 until msgBatch)
        lastbatchvalues = lastbatchvalues ++ List(valuesProposed.dequeue())
      ...
      | SEND Propose(ViewValueMsg(view,instance,lastbatchvalues))
...
```

DSL code 12: Modifications to MultiPaxos to get BatchiPaxos

Similarly to MultiPaxos, adding Batching optimization to the code took little time, with around 5% increase the number of lines and the implementation was expressive enough for the developer to understand the meaning at a glance.

5.2 Expresiveness

Our algorithm example, Paxos, is a distributed algorithm usually non trivial to implement, even for high level languages such as Java. We can state, as user experience, that when developing Paxos the translation from pseudo-code to code was pretty straightforward, off course there were details overlooked and not present in the algorithm but in general it was fast. What took more time to implement was the bootstrapping, deploying and the testing, for which Clients, Servers, Loggers and scripts were created.

Comparing the pseudo-code and the implementation we can conclude that it does not grow exponentially in number of lines, in fact the events implementation plus *preparePhase* and *proposePhase* methods can still fit into approximately 60 lines and one page.

The implementation also exhibits the ability to mix DSL and Scala code, specially inside actions, where we notice the need for conditionals, loops and collection manipulation. Furthermore, it is expressive enough to understand the purpose of the algorithm at a glance. And comparing with the Java implementation is all not only concise but also concentrated in just one scala class.

5.3 Performance

We also evaluated the performance of this implementation with the aim to see if the implementation using the DSL and most of all Akka would incur into too much overhead and to validate the DSL as capable of reproducing the same conclusions for Paxos with Batching.

The experiments were carried out on in similar settings and circumstances than JPaxos. A crash-stop model was assumed, using three machines with Pentium III 850 MHz processors, 512 MB of memory, and connected through a 100

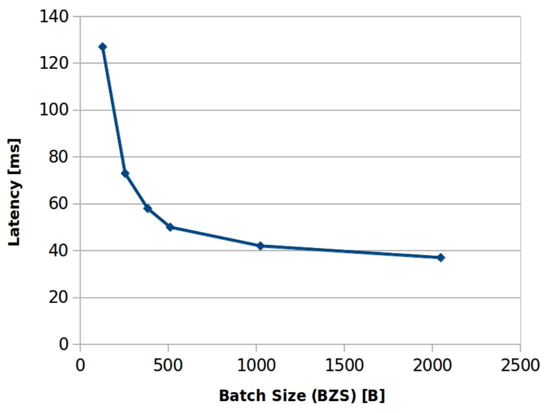

Figure 8: Clients latency in BatchiPaxos

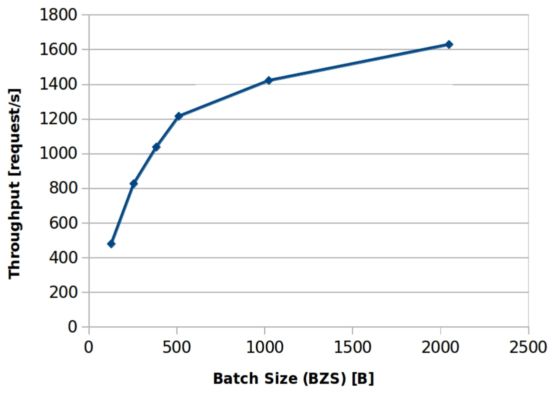

Figure 9: Clients throughput in BatchiPaxos

Mbps Ethernet cluster. Three Paxos actors would run in an ActorSystem of a replica machine. Then a client benchmark running in a fourth machine would create 60 clients and stop them after a 180 seconds run. Clients send requests of a configured S KB size each time they receive answer. Using a statistic library they save in a log the throughput calculated by the number of replies received over the time difference between the last and first ones, along with the mean and standard deviation, maximum and minimum latency or time that took a request to be replied. The first replies up to 3 seconds (same as JPaxos) were not taken into account in order to consider JVM warm-up.

Client logs where processed and using a normal distribution fit with 95% confidence interval the mean and standard deviation of all the clients was estimated. Figures 8 and 9 show the results for *BatchiPaxos*, or Paxos with batching, with request of size 128 B and batch size from 128 B up to 2048 B.

Similarly to JPaxos, we can conclude by the graphs that batching is important for small requests: throughput increases in a logarithmic fashion and latency decreases likewise.

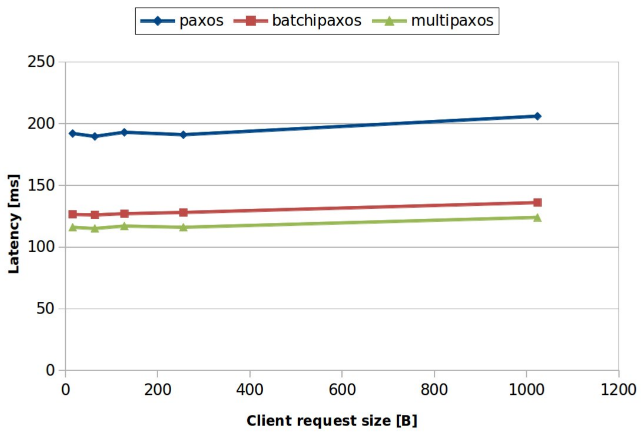

Figure 10: Clients latency comparison between MultiPaxos, BatchiPaxos and Paxos

A comparison between *Paxos*, *MultiPaxos* and *BatchiPaxos* was made similarly, this time varying the size of client requests from 16 to 1024 B. Results showed in 10 and 11 are processed the same way as for the two previous graphs.

As expected, these figures show that while there is little difference between MultiPaxos and BatchiPaxos (due to an increase in processing messages and the size of them), normal Paxos is almost twice slower than the other two, logically because there are almost twice more *preparePhase*. Another visible observation is that the size of clients request barely modifies the latency or the throughput, with what we confirm, together with batching, that the main limiting in the network is not the size of messages passed but the quantity of them.

As a general conclusion for performance evaluation we draw that the DSL allows to develop a complex algorithm in a high level way very close to pseudo-code. At the same time we are able to make similar conclusions about the performance that correspond to other implementations, in this case JPaxos. As for the performance results, maximum throughput for JPaxos reaches 12K of requests per second [15] while ours with fifteen times less clients goes up to half near 1630 requests per second. Taking into account that JPaxos is a highly optimized work and the implementation of the DSL version of Paxos took one week and even less the migration to MultiPaxos or BatchiPaxos the performance comes to second priority, due to time constraints.

In any case the goal of this project was not performance oriented, in the sense that no comparison between technologies has been studied, nor even the optimizations that JPaxos has such as serialization and careful management of threads were included this implementation. However, it would be interesting to further work in this subject in the future, along with implementing our own message passing layer. What is clear in these respects is that the DSL itself does not incur in overhead with this type of algorithms, because changes and

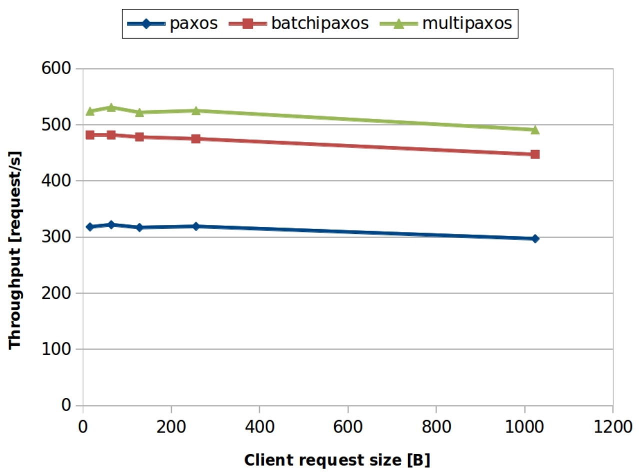

Figure 11: Clients throughput comparison between MultiPaxos, BatchiPaxos and Paxos

translations to the code are made during compile time and the processing in runtime is negligible.

6 Related work

In this section we describe frameworks, libraries, tools and programming languages that facilitate the development of distributed algorithms and systems.

While Splay [18] shares similar goals with our project in terms of expressiveness and resemblance to pseudo-code, the authors cover a wider and at the same time lower level range of usages, not specific to distributed algorithms. For instance, it has been experimented and validated with distributed systems like Chord, Pastry and BitTorrent. It also focus on providing testing and testbeds experimental facilities like churn management. Their implementation includes a central controller and a shared database between processes. In our approach the master is only used for the bootstrapping and if desired the starting point of running, furthermore the DSL actors by default are not linked to a particular process or thread, which avoids blocking.

A widely known language for distributed systems is Mace [13], which helps low level networking for distributed applications. It can be also considered a domain-specific language of type external because it fully compiles to C++ runnable code. It is a very complete system, providing failure detection, its programs can be model-checked and debugged to test safety and liveness, programmers can use existing tools to debug and profile to further analyze their systems. It has proved to be useful to researchers but in a low level kind of programming and not specific only to distributed algorithms. The way of building applications uses layer approach with upcalls and downcalls, an abstraction for events is present, and it uses a state machine approach. Although its performance is good, in the code there are too many low level details to reason directly about an algorithm like Paxos.

Neko [25] shares a similar objective to our DSL in the sense that is more focused on group communication algorithms and it is implemented in a high level language Java. However its host language does not allow the user to be expressive enough: one would still write useless code lines for example to define methods that are not really used. The environment is based on a stochastic model, the authors provide a NekoThread and NekoProcess to help controlling threading in Java along wiht support for bootstrapping.

Continuing with distributed algorithms, Mommie [19] provides an abstraction for identifying invariants or liveness properties, optimizations can be applied automatically maintaining those properties guaranties. The paper validates the work with an algorithm very similar to Paxos trying optimizations on top of it. Again, the approach is more state-machine like, they also provide an abstraction for quorum and transitions between states. In general is more focused on the verification of properties and the language may not totally intuitive to distributed algorithms.

In the network layer we find BFT protocols under fire [23], presenting a simulation environment for byzantine protocols. It includes a P2 compiler to go from pseudo-code to executable in a P2 engine, their language is designed in terms of declarative rules, closer to Lisp and logic. Another interesting work is

[9] in which the authors define a whole programming language for distributed systems with an automaton model, with the aim of reaching mathematicians, including logic signs and a formal mathematical model in their syntax.

Among the existing frameworks available for developing distributed applications, Akka stands out as a very promising and widely used platform, built on top of Scala and Java. Akka is targeted for building concurrent, distributed event-driven applications, providing a set of APIs, tools and libraries to the developer. While this simplifies implementing distributed applications in general, message passing in Akka is not expressive enough for users implementing distributed algorithms. Because its scope is very generic, it leads users to repeatedly implement some functionalities and behaviors that are common in most of these algorithms. Examples of these are the notions of triggering events, messages, algorithm layering, and verification of conditions in messages.

7 Conclusion

Nowadays programmer productivity has a preponderant value for both the industry and academia. Concretely, in distributed systems there is a gap between research algorithms and real executable implementations, which may explode into thousands of lines of code. Furthermore the deployment, validation and verification of these implementations are not easy tasks and usually require more time and attention from the developer. We presented DISTAL, a new Domain-Specific Language for implementing distributed algorithms that provides a very high level and intuitive syntax on top of Scala.

This DSL has been validated with Paxos, its variant MultiPaxos and an optimization BatchiPaxos. We showed that the DSL library allows the developer to easily and rapidly modify code, changing from one algorithm to the other, using abstractions such as Event triggers and Messages that resemble their pseudo-code counterparts. Through experimentation, we evidenced that the execution of these algorithms with our library, follows a similar behavior than in other implementations like JPaxos. Although the performance evaluation exhibits an overhead, which might be subject to study for future work, there were no optimizations made to the code or the message passing settings. Besides, development time is greatly reduced, while expressiveness is maintained without exploding in terms of lines of code (from 150 of the algorithm pseudo-code to 183 its implementation), in contrast to several thousands of lines in languages like C++ [5].

The main benefits of this project are not only improving development productivity and clarity but also reducing errors while translating algorithms pseudo code to our language. DISTAL is the result of an analysis and modeling of distributed algorithms, based on events and messages, not state machine like most of existing approaches in the state of the art. This makes the transition form algorithms to programming code very simple, unlike other languages. Moreover, DISTAL emphasizes on the notion of reuse. Nowadays most of internal DSLs define a small but complete language with its own syntax and interpreter which requires to redefine, for latter translation, features already present in the host language. One of the novelties of this DSL lies in the ability of inserting and mixing native Scala code to take advantage of its useful features. Finally, we showed in the the experiments that a well known and complex distributed algorithm such as Paxos can be easily implemented using DISTAL, while preserving the algorithm behavior as if we were using other implementations.

There exist areas that can be the subject of further study, starting with a more in depth comparison between DSL, Akka, Scala and Java only implementations, continuing with testing different settings for message passing, and profiling. Another useful tool can be one made to verify in an automatic way the correctness and performance of the algorithms developed.

References

[1] Yanif Ahmad. *Pulse: database support for efficient query processing of temporal polynomial models.* PhD thesis, Providence, RI, USA, 2009. AAI3377093.

[2] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing paxos. *ACM Sigact News*, 34(1):47–67, 2003.

[3] C. Cachin. Yet another visit to paxos. 2010.

[4] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming.* Springer-Verlag New York Inc, 2011.

[5] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live-an engineering perspective (2006 invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing-PODC*, volume 7, 2007.

[6] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[7] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

[8] M. Fowler and Parsons R. *Domain-Specific Languages.* Addison-Wesley, 2011.

[9] S.J. Garland and N.A. Lynch. Using i/o automata for developing distributed systems. *Foundations of Component-Based Systems*, 13:285312, 2000.

[10] D. Ghosh. *DSLs in action.* Manning Publications, 2011.

[11] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. 1994.

[12] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, et al. Green-marl: A dsl for easy and efficient graph analysis. *ACM Transactions on Graphics*, 2009.

[13] C.E. Killian, J.W. Anderson, R. Braud, R. Jhala, and A.M. Vahdat. Mace: language support for building distributed systems. *ACM SIGPLAN Notices*, 42(6):179–188, 2007.

[14] J. Kirsch and Y. Amir. Paxos for system builders. Technical report, Technical Report CNDS-2008-2, Johns Hopkins University, 2008.

[15] J. Kończak, N. Santos, T. Żurkowski, P.T. Wojciechowski, and A. Schiper. Jpaxos: State machine replication based on the paxos protocol. 2011.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[17] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[18] L. Leonini, É. Rivière, and P. Felber. Splay: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 185–198. USENIX Association, 2009.

[19] P. Maniatis, M. Dietz, and C. Papamanthou. Mommie knows best: systematic optimizations for verifiable distributed algorithms. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 30–30. USENIX Association, 2011.

[20] Z. Milosevic, M. Hutle, and A. Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 235–244. IEEE, 2011.

[21] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Communications of the ACM*, 55(6):121–130, 2012.

[22] N. Santos, M. Hutle, and A. Schiper. Latency-aware leader election. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1056–1061. ACM, 2009.

[23] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. Bft protocols under fire. NSDI, 2008.

[24] A.K. Sujeeth, H. Lee, K.J. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML*, 2011.

[25] P. Urban, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Information Networking, 2001. Proceedings. 15th International Conference on*, pages 503–511. IEEE, 2001.

[26] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.