A Distribution Definition Language for the Automated Distribution of Java Objects

Paul Soule Tom Carnduff Stuart Lewis

Faculty of Advanced Technology
University of Glamorgan
Pontypridd Wales
U.K. CF37 1DL
{psoule,tcarnduf,sflewis}@glam.ac.uk

Abstract

Distributed applications are difficult to write. Programmers need to adhere to specific distributed systems programming conventions and frameworks, which makes distributed systems development complex and error prone and ties the resultant application to the distributed system because the applications code is tangled with the crosscutting concern *distribution*.

We introduce a simple high level domain specific aspect language we call a Distribution Definition Language (DDL), which describes the classes and methods of an existing application that are to be made remote, the distributed system to use to make them remote, and the recovery mechanism to use in the event of a remote error. The DDL is used by the RemoteJ compiler / generator to generate the distributed system specific code and apply it to components using bytecode manipulation techniques.

We describe the language and its features and show that a distribution definition language can be used to significantly simplify distributed systems development and improve software reuse.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, compilers

General Terms Languages, Experimentation.

Keywords Distributed systems, RemoteJ, domain specific aspect languages, aspect-oriented programming, code generation.

1. Introduction

Existing mainstream programming languages, such as Java, do not provide language support for distribution. Rather, programmers must rely on object-oriented distribution frameworks, such as RMI and Jini, to provide distribution support. Although highly successful, the cost of using these frameworks is that the resultant code is tied to the framework. Object-oriented frameworks in general, and distribution frameworks in particular, can therefore be considered crosscutting in nature because the frameworks code, either via inheritance or containment, is scattered throughout the applications code thereby tying the application to the framework.

This is a particular concern in distributed systems because distribution frameworks impose a large code overhead due to the requirements distributed systems impose, such as the need to catch distribution specific exceptions, locating and binding to distributed objects, locating another server in the event the current server becomes unavailable, and adhering to programming conventions dictated by the framework, such as implementing framework specific interfaces. Consequently, developing distributed applications is complex and error prone and results in application components tied to the distribution framework, which cannot be easily reused outside the application. Distribution is therefore considered a crosscutting concern and consequently a prime candidate for an aspect-oriented [7] approach.

Several attempts have been made to apply distribution aspects to existing Java code. These attempts typically target a single distribution protocol, RMI, and either generate AspectJ [6] code [1, 12, 16], use a domain specific language [8, 9], extend Java [11], or extend the AspectJ language to provide distribution [10]. While RMI is the most widely used distribution protocol in Java systems and is used as the protocol for Enterprise JavaBeans (EJB), Jini and JavaSpaces, there are a number of other distributed systems, such as CORBA, JMS, SOAP, HTTP, Java sockets etc. that Java programmers may choose to use and indeed may have to use to solve a particular integration problem. Developers therefore have a large choice of protocols, each with its own framework and possibly different programming convention. This significantly complicates distributed systems development.

We introduce the concept of a distribution definition language, a simple high level domain specific aspect language, which generalises distributed systems development by describing the classes and methods to be made remote, the distributed protocol to use to make them remote, and the method used to recover from a remote error. The DDL is used by the RemoteJ compiler / generator, which uses bytecode manipulation and generation techniques to provide a distributed version of the application while retaining existing classes for reuse in other distributed or non-distributed applications.

By generalising and modularising the distribution concern, the use of a DDL provides a method of developing distributed applications that is significantly simplified, allows multiple protocols to be supported for the same code base, and allows the same code to be used in both a distributed and non-distributed application thereby improving software reuse.

2. RemoteJ

RemoteJ is a compiler / generator that is used to apply distribution to existing Java components using instructions contained in a distribution definition language format file. The salient features of the RemoteJ system are:

Workshop DSAL '07 March 12, 2007 Vancouver, British Columbia, Canada.

- A Distribution Definition Language used to describe the classes and methods to be made remote, the protocol to use to make them remote, and the strategy used to recover from remote errors.
- A recursive descent parser used to parse and validate the DDL.
- A code generator used to generate generic distribution and recovery code.
- A protocol abstraction used as the basis for one or more pluggable protocol generators.
- One or more protocol implementations used to generate distribution specific code.
- A bytecode rewriter to alter existing bytecode in a non-destructive way.

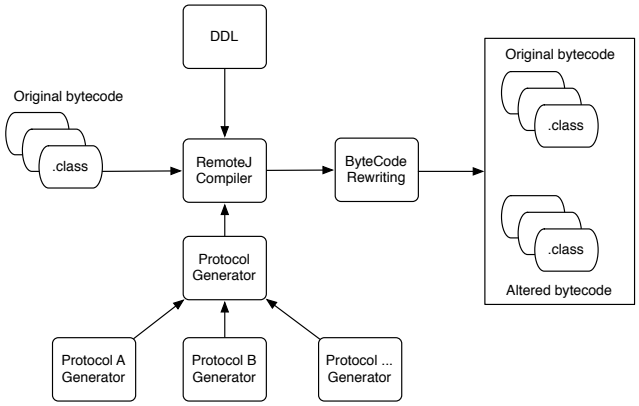

Figure 1. System Overview

The DDL is read by the RemoteJ compiler / generator, which alters the bytecode for the classes and methods described in the DDL and writes the altered bytecode to a directory structure stipulated by the DDL's service statement thereby ensuring that the original bytecode remains unaltered.

2.1 The Distribution Definition Language

The DDL is a simple language, based on a Java-like syntax, used to describe classes and their methods to be made remote, the protocol to be used, and the action to take in the event of an error. The DDL is designed to support any number of protocols and recovery strategies in the same source file thereby allowing a single source file to be used to apply distribution to any number of class files from the same source file. The RemoteJ DDL language is illustrated in EBNF format in Figure 4 and a simple DDL sample is illustrated in Figure 2, which will be used as the basis for this discussion.

2.1.1 Comments

Comments in the DDL are the same as for the Java language.

end of line
Line comments start with // and end at the end of the line

multi line
Multi line comments start with /* and end with */

2.1.2 Keywords

The DDL reserves the following keywords, which therefore cannot be used as identifiers.

```
import uk.ac.glam.*;

service TestService {

   recovery remoteError (RemoteException e) {
      System.out.println("Exception: " +
            e.getMessage());
   }

   protocol : rest {
      options {
         baseURI = "/myserver";
         portNumber = 1234;
         SSL = false;
       }

       export * Account.* (String, String, int) {
          recovery = remoteError;
       }
   }

   protocol : rmi {
      options {
          registryName = "RMITestServer";
          registryHost = "localhost";
          registryPort = 1099;
          runEmbeddedRegistry = true;
       }

       export * Address.* (String, String) {
          recovery = nextServer;
       }
   }
}
```

Figure 2. Sample RemoteJ DDL Code

Keyword:
import | service | recovery | protocol | rest |
rmi | options | export | nextServer | abort | continue

As well as the above, there are a number of reserved works that are dependent on the protocol that has been selected[1]. These keywords are:

for the rmi protocol:
registryName | registryHost | registryPort |
runEmbeddedRegistry

for the rest protocol:
baseURI | portNumber | SSL

2.1.3 Import Statements

The DDL may contain any number of import statements. As in the Java language, this is used to avoid having to use fully qualified class names when referring to Java classes. Import statements may only appear at the beginning of the DDL file.

2.1.4 Service Statement

The service statement is used to describe one or more protocols, and associated classes, and one or more recovery statements. The name used for the service must be the same as the name of the DDL file with the extension ddl, for example the DDL described in Figure 2 must be contained in the file `TestService.ddl`.

[1] Note that additional protocols will introduce additional keywords.

The service name is used as the directory name for generated classes prefixed, by default, by either 'client', for client classes, or 'server' for server classes.

2.1.5 Service Recovery Statements

Service recovery statements are used to provide the code to be called in the event of distribution exceptions. Any valid Java code can be stipulated, which allows a great deal of control over the recovery mechanism as the programmer is free to provide any recovery implementation not explicitly supported in the language providing it can be found by the RemoteJ compiler / generator (i.e. in the compiler's CLASSPATH).

Recovery statements have access to the context of the remote method call via RemoteJ's internal `Transfer` object. This object contains the remote server name, the class and method that was called, and the method's parameters and return type.

Recovery statements are defined with the keyword **recovery** followed by the name of the recovery statement and the exception to be caught as illustrated below.

```
recovery remoteError (RemoteException e) {
    System.out.println("Exception: " +
          e.getMessage());
    System.out.println("Host : " +
         transfer.getCurrentHost());
    System.out.println("Failed method call : " +
         transfer.getMethod());
}
```

In the above example, `RemoteException` is used as the exception type. The protocols supported by the RemoteJ DDL may, however, not use `RemoteException` to indicate that an error has occurred. In these cases, the exception type provided by the protocol may be used. In the cases where error codes are used in place of exceptions, the protocol implementation is responsible for providing an exception hierarchy and appropriate mapping between error codes and exceptions.

Any number of recovery statements may be provided but will only be invoked if called by one or more recovery statements contained in the export statement.

In addition, the DDL supports three additional statements that may be used to aid recovery:

nextServer: The protocol implementation should attempt to recover from a remote error by finding an additional server.

abort: The protocol implementation should stop in the event of a remote error.

continue: The protocol implementation should ignore remote errors. This statement is provided for completeness.

2.1.6 Protocol Statements

The protocol statement is used to describe the protocol to be used, the protocol options, the classes and associated methods to be altered to use the protocol, and the recovery strategy to be used.

In the example in Figure 2, two protocols have been specified, the rmi protocol and the rest protocol. There may be any number of protocol statements and there may be more than one protocol statement for the same protocol. Each protocol statement must contain one or more export statements.

2.1.7 Options Statements

As can be expected, different protocols may have different protocol options and these options are stipulated in the options statement contained in the protocol statement. As there may be more than one protocol in a single application (or perhaps the same protocol with different options, for example), any number of protocol and associated option statements may be declared.

2.1.8 Export Statements

The export statement contains the class and associated methods that are to become distributed using the protocol stipulated in the protocol statement. In addition, the recovery strategy may be stipulated for those exported methods.

The export statement supports the use of the asterisk wildcard character, which is used as follows:

The statement

export ∗ Address.∗ (String , String);

stipulates that all methods in the class `Address` (Address.*) with two parameters of the type `String` that have any type as a return statement (*) are selected.

A compiler error is generated if a method is matched by more than one export statement.

3. RemoteJ Compiler

The RemoteJ compiler is a simple three-phase compiler / generator that is used to apply distribution to existing bytecode using instructions contained in a DDL file.

Any number of back end code generators are supported thereby allowing the compiler to be extended to support additional protocols. Code generation support is provided by the abstract base class, `Protocol`, which contains generalised protocol support functions and defers specific protocol implementation to concrete subclasses based on the protocol selected in the DDL using an implementation of the template method design pattern [4].

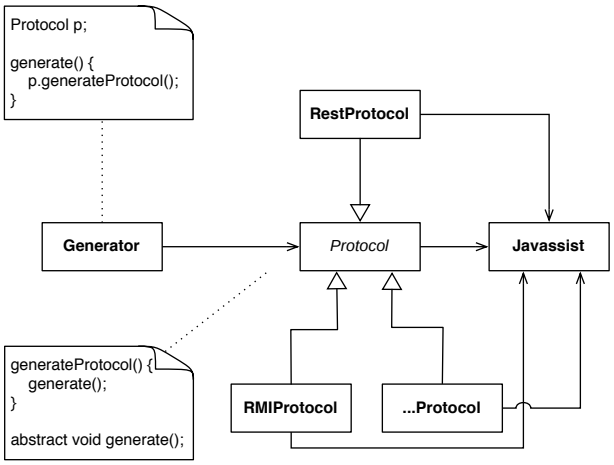

Figure 3. Extendible code generation support

The compilers syntactic analysis phase is implemented as a recursive descent parser and is therefore easy to extend to support additional protocols. The abstract syntax tree (AST) generated by the syntactic analysis phase is used by the contextual analysis phase to ensure the DDL conforms to the DDLs contextual constraints. This phase is implemented using the Visitor design pattern [4].

In contrast to other code manipulation systems that generate AspectJ code, the RemoteJ compiler manipulates bytecode directly using Javassist [2], a simple bytecode manipulation library. This allows the manipulated code to be placed in a different directory structure thereby leaving the original bytecode intact.

4. Discussion

Remote Procedure Calls (RPC) were designed to overcome the difficulties of distributed systems development where developers were

```
ddl             ::= (ImportList)* Service
ImportList      ::= SingleImport (SingleImport)*
SingleImport    ::= "import" Imports Semi
Imports         ::= Identifier (Dot Identifier | WildCard)*
Service         ::= "service" Identifier LeftCurly
                    StatementList RightCurly
StatementList   ::= (RecoveryList)* (ProtocolList)+
RecoveryList    ::= RecoveryStatement (RecoveryStmt)*

RecoveryStmt    ::= "recovery" RecoveryName LeftBracket
                    ClassName Variable RightBracket
                    LeftCurly AnyJavaCode RightCurly
ProtocolList    ::= ProtocolStmt (ProtocolStmt)*
ProtocolStmt    ::= "protocol" ProtocolList Colon LeftCurly
                    Options (ExportStatement)+ RightCurly
ProtocolList    ::= "rmi" | "rest"
Options         ::= "options" LeftCurly (RmiOptions)+
                    | (RestOptions)+ RightCurly

RmiOptions      ::= RegistryName | RegistryHost
                    | RegistryPort | Embedded
RegistryName    ::= "registryName" Equals Quote Identifier
                    Quote Semi
RegistryHost    ::= "registryHost" Equals Hostname
                    | IpAddress Semi
RegistryPort    ::= "registryPort" Equals IntegerLiteral Semi
HostName        ::= Quote Identifier Quote
IpAddress       ::= IntLiteral Dot IntLiteral Dot
                    IntLiteral Dot IntLiteral
Embedded        ::= "runEmbeddedRegistry" Equals "true"
                    | "false" Semi

RestOptions     ::= BaseURI | PortNumber | SSL Semi
BaseURI         ::= "baseURI" Equals Quote
                    Identifier Quote
PortNumber      ::= "portNumber" Equals IntegerLiteral
SSL             ::= "SSL" Equals "true" | "false"
exportStatement ::= "export" ReturnValue ExportName
                    LeftBracket (Parameter)?
                    (Comma Parameter)* RightBracket
                    LeftCurly RecoveryType RightCurly
RecoveryType    ::= "recovery" Equals RecoveryOption Semi
RecoveryOption  ::= RecoveryName | "continue" | "abort"
                    | "nextServer"
RecoveryName    ::= Identifier
ClassName       ::= Identifier (Dot Identifier)*
Parameter       ::= ClassName
ExportName      ::= Identifier (Dot Identifier | WildCard)*
Variable        ::= Identifier
RecoveryType    ::= Identifier
IntegerLiteral  ::= 0..255
Identifier      ::= ('a'..'z' | 'A'..'Z')+ (IntegerLiteral)*
Comma           ::= ","
Dot             ::= "."
Semi            ::= ";"
LeftCurly       ::= "{"
RightCurly      ::= "}"
LeftBracket     ::= "("
RightBracket    ::= ")"
Colon           ::= ":"
Equals          ::= "="
Quote           ::= ""
WildCard        ::= "*"
```

Figure 4. RemoteJ DDL Language (simplified)

required to deal with low-level details such as network connections, protocol handling, data representation between different architectures, both partial and 'hard' failures, reassembly of data packets and various other issues. RPCs were designed to behave the same as local procedure calls by masking the difference between local and remote procedure calls so that, to the developer, local and remote procedure calls were essentially equivalent.

RPC systems introduced the concept of an Interface Definition Language (IDL), a language that defines procedures that are to be distributed. The IDL language file is read by an IDL compiler, which generates the stubs and skeletons, for a specific protocol, that the developer uses to implement the distributed concern.

RPCs were designed for procedural languages and do not provide object-oriented features, such as polymorphism, because RPCs only allow for a static representation of data. Modula-3 network objects [3] and, recently, Java's Remote Method Invocation (RMI) paradigm extended the RPC concept to distributed object-oriented systems that allows for the dynamic representation of polymorphic data.

However, the developers of RMI argue that the RPC concept of masking the differences between local and remote procedure calls is flawed because there are fundamental differences between the interactions of distributed objects and the interactions of non-distributed objects [17]. Consequently the framework provided by RMI requires that the developer be aware of remote objects and remote errors that may occur while interacting with remote objects. This awareness manifests itself in the need for developers to adhere to the RMI framework and to ensure that remote methods throw the RMI specific exception `java.rmi.RemoteException`.

While the RMI designers may well be correct in insisting that programmers are aware of distribution concerns, the implementation of this requirement leads to code that is polluted with the crosscutting concern *distribution* because the distribution concern crosscuts the application making reuse of application components difficult, if not impossible. This close coupling between frameworks and application code is not unique to RMI, it is inherent in all object-oriented applications that use frameworks. Frameworks may therefore be considered crosscutting in nature because a frameworks code is scattered throughout an applications code, either by inheritance or containment, thereby making reuse outside the frameworks domain difficult.

Aspect-oriented programming [7] has been suggested as a way of modularising crosscutting concerns and applying them to existing code in an oblivious manner and distribution is considered a typical candidate for an AOP application as distribution code can be designed as an aspect, which separates it from the actual application.

There have been a number of different approaches to the implementation of distribution as aspects in Java, such as frameworks e.g. [1], extensions to Java e.g. [10], or the use of a general aspect-oriented language e.g. [12]. Our approach is to raise the level of abstraction by the use of a high level domain specific language that allows programmers to declare the classes and methods of those classes that they would like to be distributed, and the protocol to use. The DDL has no concept of pointcuts or join points but instead relies on a high level approach that eliminates the need for the developer to be aware of aspect-oriented concepts, techniques, and implementation methods.

In common with most aspect-oriented language implementations, RemoteJ relies on bytecode rewriting to implement distribution. However, in contrast to current aspect-oriented languages and frameworks, RemoteJ does not overwrite the applications bytecode, rather distribution is applied to a copy of the classes thereby preserving the original code so that the same application may be run distributed or not by simply altering the systems CLASSPATH variable.

4.1 Advantages and Limitations

The RemoteJ system currently has a number of limitations, which we outline below.

Object serialization. There are a number of issues to the automated distribution of Java objects. The first concerns the concept of obliviousness. We agree with Soares et al. [12] that systems that are to be distributed need to be aware of the possibility of becoming distributed. The overriding reason for this is that, for most Java protocols, parameter or return value classes must implement the `Serializable` interface so that the object may be converted into a byte stream and transmitted across the wire (copy-by-value). If a class does not implement the `Serializable` interface it cannot be converted into a byte stream and, at least with the RMI protocol, needs to implement the `Remote` interface if it is to be used in a remote method call, which allows the object to be accessed by a remote procedure call from the server to the client. Indeed, this is the solution proposed by Soares et al. [12]. We believe, however, that this is not generic enough for our solution, which supports multiple protocols and is anyway likely to lead to performance issues. A simple solution may be to simply force a class to be `Serializable`. However, this cannot work for all classes, particularly if the class, or any containing class, cannot be serialized because it contains an open file or socket, for example. We believe that the developer needs to be aware of the possibility of distribution and consequently needs to ensure that classes that are passed as parameters or return values implement the `Serializable` interface. The RemoteJ compiler therefore generates an error if parameter and return classes are not `Serializable`.

Local and remote object consistency. In remote object protocols, such as RMI, changes applied to remote objects are not reflected in local objects. The NRMI system and GOTECH framework [16] overcomes this for the RMI protocol. However, in our system, this needs to be overcome in a generalised way so that it can be supported for all protocols that RemoteJ may support. We plan to provide this feature in a future release.

Thread management. Java distribution protocols do not support thread coordination so that Java language synchronization operations (*synchronized, wait, notify etc.*) are not propagated across the network. Tilevich and Smaragdakis [14] discuss this issue in some detail along with a proposed solution for the automated distribution of Java objects in the J-Orchestra system [15]. However, their solution does not address all situations and, as with object serialization detailed above, we believe this issue is best left to the programmer to resolve. The RemoteJ compiler will therefore issue a warning if a method to be distributed uses one of the thread co-ordination operands.

We believe the above limitations are insignificant compared to the advantages the concept of a distribution definition language offers. These advantages can be summarised as:

Simplicity. The ability for the developer to apply distribution to existing code without having to learn aspect-oriented concepts, distribution frameworks, or protocols.

Error handling. Centralised error handling and recovery capabilities, including automated recovery e.g. automatically connecting to an alternate server if the current server becomes unavailable.

Multiple protocol support. The ability to support multiple protocols for the same code base.

Reuse. Components can be used as distributed components or local components thereby improving reuse for those components as the RemoteJ system does not alter the existing bytecode, rather it produces a copy of the classes with the distribution concern applied. This is in contrast to current distributed programming practises where the component is tied to the distributed system through the need to adhere to a distribution specific framework.

As well as the advantages detailed above, the concept of a domain specific language for the application of crosscutting concerns is a powerful concept that may be applied to other crosscutting concerns besides distribution. For example a persistence definition language may be used to apply the persistence crosscutting concern to existing applications.

5. Related Work

Although many domain specific languages (e.g. [5]) have been proposed to aid distributed programming, few have used aspect-oriented concepts to apply the distribution concern to existing applications. The closest work to ours thus far is the D and AWED languages.

The D language [8] was the first language to provide explicit support for distribution. D consists of the COOL synchronisation sub-language and RIDL, a sub-language for the definition of remote interfaces. AWED [9] is a comprehensive aspect language for distribution which provides remote pointcuts, distributed advice, and distributed aspects and is implemented by extensions to the JAsCo [13] AOP framework.

Both D and AWED use a much lower level approach than RemoteJ, which, by dispersing with the AOP notions of pointcuts, advice, and aspects in favour of a distribution definition language, is at a much higher level of abstraction. Consequently RemoteJ is much easier to use than previous approaches. In addition, RemoteJ supports centralised exception management, which is not supported by either D or AWED.

A number of systems, such as [1, 12, 16], use the general purpose AspectJ [6] language to apply distribution concerns to existing applications. These approaches target a single protocol, typically RMI, and generate AspectJ code to implement distribution.

JavaParty [11] and DJcutter [10] use a language based approach and supply Java language extensions to provide explicit support for distribution. Again these systems target the RMI protocol.

The J-Orchestra system [15] is an automatic partitioning system for Java. J-Orchestra uses bytecode rewriting to apply distribution and claims to be able to partition any Java application and allow any application object to be placed on any machine, regardless of how the application objects interact. Users interact with the J-Orchestra system using XML files, and a GUI is currently under development. However, J-Orchestra does not have the concept of a domain specific language and targets a specific protocol, again RMI.

We believe the RemoteJ system is the first automated distribution system to provide the concept of a Distribution Definition Language that targets multiple protocols and that provides a central recovery mechanism for distribution errors.

6. Conclusion and Future Work

In this paper, we have presented the RemoteJ system, a system that consists of a compiler / generator to apply distribution to existing applications using information contained in a high level domain specific language we call a Distribution Definition Language. RemoteJ uses an aspect-oriented approach to apply the distribution concern to existing code by altering bytecode to apply the distribution concern. We believe this concept significantly simplifies distributed systems development and improves software reuse.

RemoteJ currently has a number of limitations and issues, such as protocol agnostic local and remote object consistency and thread co-ordination, that needs to be fully explored. Additional language constructs, such as clustering and replication, will improve the capability and sophistication of the system and further protocol support will further refine the language.

References

[1] M. Ceccato and P. Tonella. Adding Distribution to Existing Applications by Means of Aspect Oriented Programming. In *Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, 2004, pp. 107-116.

[2] S. Chiba and M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of the 2nd international conference on Generative Programming and Component Engineering (GPCE '03)*, Springer-Verlag, 2003, pp. 364-376.

[3] D. Evers and P. Robinson. Modula-3 network objects over ANSA: heterogeneous object-based RPC in a modern systems programming language In *Proceedings of the 5th workshop on ACM SIGOPS European workshop*, Mont Saint-Michel, France, 1992, pp 1–5.

[4] E. Gamma, R. Helm, R. E. Johnson and J. Vissides. *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[5] E. B. Henri. *Orca: a language for distributed programming*, SIGPLAN Not., 25 (1990), pp. 17–24.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold. *An Overview of AspectJ*, Lecture Notes in Computer Science, 2072 (2001), pp. 327–355.

[7] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, Berlin, Heidelberg, and New York Springer-Verlag, 1997, pp. 220–242.

[8] C. V. Lopes and G. Kiczales. *D: A Language Framework for Distributed Programming*, 1997.

[9] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine and D. Suvée Explicitly distributed AOP using AWED. In *Proceedings of the 5th international conference on Aspect-oriented software development*, Bonn, Germany, 2006, pp. 51–62.

[10] M. Nishizawa, S. Chiba and M. Tatsubori. Remote pointcut: a language construct for distributed AOP. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, Lancaster, UK, 2004, pp. 7–15.

[11] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. In *Concurrency: Practice & Experience*, 9 (1997), pp. 1225–1242.

[12] S. Soares, E. Laureano and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, Seattle, Washington, USA, 2002, pp. 174–190.

[13] D. Suvée, W. Vanderperren, and V. Jonkers JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003, pp. 21–29.

[14] E. Tilevich and Y. Smaragdakis. Portable and Efficient Distributed Threads for Java. In *ACM/IFIP/USENIX 5th International Middleware Conference (Middleware '04)*, Toronto, Ontario, Canada, 2004.

[15] E. Tilevich and S. Urbanski. J-Orchestra: Automatic Java Application Partitioning. In *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, 2002.

[16] E. Tilevich, S. Urbanski, Y. Smaragdakis and M. Fleury. Aspectizing Server-Side Distribution. In *21st IEEE/ACM International Conference on Automated Software Engineering*, Tokyo, Japan, 2003.

[17] J. Waldo, G. Wyant, A. Wollrath and S. Kendall. A Note on Distributed Computing. *Sun Microsystems Laboratories*, Inc., 1994.