An Aspect-Oriented Communication Middleware System

Marco Tulio de Oliveira Valente, Fabio Tirelo, Diana Campos Leao, Rodrigo Palhares Silva

Department of Computer Science,
Catholic University of Minas Gerais, Brazil
{mtov,ftirelo}@pucminas.br

Abstract. This paper describes a Java-based communication middleware, called AspectJRMI, that applies aspect-oriented programming concepts to achieve the following requirements: (1) modular implementation of its features, including those with a crosscutting behavior; (2) high degree of configurability and adaptability; (3) performance similar to conventional object-oriented communication middleware systems, such as CORBA and Java RMI. In AspectJRMI, users may explicitly select the features provided by the middleware infrastructure, according to their needs. Most of these features have a crosscutting behavior, including interceptors, oneway calls, asynchronous calls, value-result parameter passing, and collocation optimizations. In this case, they are implemented as aspects. The design of AspectJRMI follows a set of principles, called horizontal decomposition, to achieve pluggability of aspects to the core middleware implementation. This paper presents the programming interface and the implementation of AspectJRMI. It also presents experimental results of its performance.

1 Introduction

Middleware systems, such as CORBA [14] and Java RMI [24], encapsulate several details inherent to distributed programming, including communication protocols, data marshalling and unmarshalling, heterogeneity, service lookup, synchronization, and failure handling. Although such systems have been proposed to make distributed programming more simple and natural, they have evolved over the years to become complex, monolithic, and heavyweight [7, 26, 28]. On the other hand, a wide range of applications use only a reduced subset of middleware features. In this case, the monolithic architecture of middleware contributes to increase the complexity, size, and the resource requirements of such systems, without providing proportional benefit.

This paper describes a Java-based communication middleware, called AspectJRMI, that applies aspect-oriented programming concepts to achieve the following requirements:

- Modular and open implementation of features, including those having a crosscutting behavior. In AspectJRMI, users can change and extend the main components of the platform.

– High degree of configurability and pluggability. At compile time, users may explicitly select the features provided by the middleware infrastructure, according to their needs.

– Performance similar to conventional object-oriented communication middleware systems, such as CORBA and Java RMI.

In order to achieve such requirements, the design of AspectJRMI follows a set of principles, called horizontal decomposition [28], that advocates the synergistic combination of objects and aspects in order to modularize middleware concerns. The components of AspectJRMI are divided in two categories: mandatory and non-mandatory. Following the guidelines proposed by horizontal decomposition method, mandatory (or core) components are those related to the main middleware functionality, i.e., components that support the implementation of transparent remote method invocations (using call-by-value, at-most-once, and synchronous semantics). As examples of mandatory components, we can mention channels, stubs, skeletons, and remote references. Moreover, in AspectJRMI users can extend or adapt the default behavior of mandatory components. For example, they can define channels that use different transport protocols or add extra functions to channels (such as encryption or compression of messages). Mandatory components were implemented using traditional techniques in object-oriented programming (such as design patterns and vertical decomposition). Particularly, the core of the system was implemented using components defined in Arcademis [17], a Java-based framework that supports the implementation of middleware architectures.

Non-mandatory components implement features that are not required in every distributed application. Most of these features have a crosscutting behavior, such as interceptors, oneway calls, asynchronous calls, value-result parameter passing, and collocation optimizations [26, 27]. For this reason, they are implemented as aspects using AspectJ [11]. An aspect compiler is used to weave the core and optional components. Moreover, the static nature of the weaving process in AspectJ contributes to keep the performance and memory footprint overhead of AspectJRMI at acceptable values.

The rest of the paper is structured as follows. Section 2 presents an overview of the main principles advocated by horizontal decomposition. Section 3 describes the core of AspectJRMI. First, the section describes the overall architecture of Arcademis and then describes the main components of the AspectJRMI core. Section 4 presents the aspects that can be woven to the core in order to support the following crosscutting features: oneway calls, asynchronous calls, call by value-result, service combinators, remote reference decorators and collocation optimizations. Section 5 describes an experimental evaluation of the size and performance overhead of AspectJRMI. Section 5 describes related work and Section 6 concludes the paper.

2 Horizontal Decomposition

Horizontal decomposition is a set of guidelines and principles that support the implementation of middleware systems with high degrees of modularity and adaptability [28]. The method proposes a solution to the *feature convolution* phenomenon in traditional middleware systems, i.e., the fact that many middleware features can not be easily plugged in and plugged out of the platform, since they crosscut the implementation of other features. Horizontal decomposition advocates the use of traditional modularization techniques, such as vertical decomposition, to implement a minimal but well-modularized core middleware system. Basically, this core should provide support to synchronous and statically defined remote invocations, using call-by-value parameter passing. Horizontal decomposition also advocates the use of aspect-oriented programming to superimpose orthogonal features in this core. A feature is considered orthogonal if both its semantics and implementation do not fit in a single component. Traditional weaving process is used to compose the core and the orthogonal features that are requested in a particular application. The method supports fine-grained customizations; particularly, features not used by an application do not impact the generated middleware platform.

3 AspectJRMI Core

The core of the system is derived from components defined in Arcademis [17], a Java-based framework that supports the implementation of customizable middleware architectures.

3.1 Arcademis Architecture

Arcademis predefines the overall architecture of middleware platforms, as described in Figure 1. In object-oriented middleware platforms, clients traditionally use intermediate components to invoke methods on remote objects. Two of these components are the stub, which exists on the client side of a distributed application, and the skeleton, which is located on the server side. The stub acts as a local proxy for the remote object, and its function is to forward to the server remote calls made by the client. The skeleton represents the invoking client to the remote object, acting as an adapter.

Besides stubs and skeletons, Arcademis defines several other components. The `invoker` is responsible for emitting remote calls, whereas its server counterpart, the `dispatcher`, is in charge of receiving and passing them to the skeleton. The `Scheduler` is used whenever necessary to sort remote calls according to their priorities. The communication layer in Arcademis is represented by a set of components that constitute the transport protocol, the serialization protocol and the middleware protocol. Connections are established by two components: the `Connector` and the `Acceptor`. Request senders and receivers provide means to assure the reliability level the middleware provides to distributed applications.

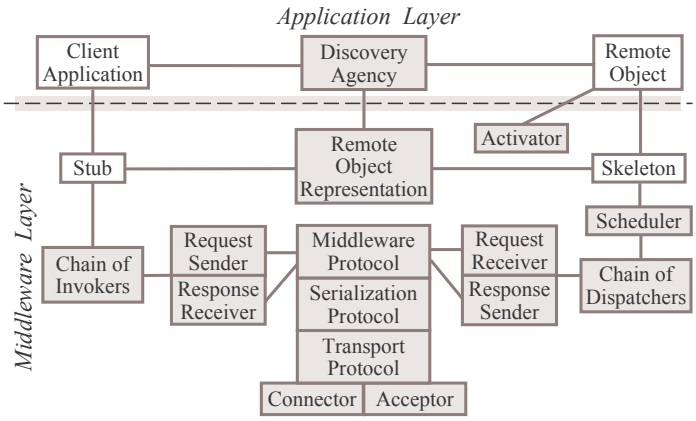

Fig. 1. Arcademis Architecture

A lookup service allows clients to discover and access remote objects. Finally, the `Activator` defines how an object is made ready for receiving remote calls.

3.2 AspectJRMI Core Components

The core of AspectJRMI is an instance of Arcademis that provides a basic remote method invocation service. The components in the core implement interfaces and abstract classes defined in Arcademis and reuse concrete classes from this system. Including Arcademis components, the core is composed by 4 interfaces, 11 abstract classes, 89 concrete classes and almost 9000 lines of Java code. The core also includes a stub/skeleton compiler.

Communication Components: The core uses TCP/IP for data transmission. It also supports a middleware protocol with four different types of messages: *call*, *return*, *ping* and *ack*. The *call* message describes a remote invocation, including its arguments and identifiers. The *return* message holds the results of remote calls. Messages *ping* and *ack* are used in order to verify if servers or clients are alive.

In the core, acceptor and connector components provide a synchronous service, meaning that the client thread remains blocked during the execution of remote calls. On the server side, a connector creates a new thread for each incoming connection. Service handler components (request sender/receiver and response receiver/sender) implement an at-most-once invocation semantics.

Client and Server Configurations: We have implemented two versions of the core of AspectJRMI. The first version has only client functionality, i.e., it has only components in charge of dispatching remote method invocations, such

as connector, invoker, request sender and stub. In the current implementation of the system the client core has 37 KB (size of all the .class files). The second version supports client and server functionality, i.e., it has components in charge of sending and receiving remote method invocations. Besides the client components, this version includes the following elements: acceptor, activator, connection server, dispatcher, skeleton and request response. The client/server core has 68 KB.

Reconfiguration of Core Components: Following the Arcademis architecture, the core of AspectJRMI has a singleton component called ORB. The ORB contains a set of factories that eases changing the implementation of a component without interfering in other modules of the system. In order to extend the behavior of a component, users should only associate a new factory to the ORB. The ORB has factories for the following components: channels, service handlers, activators, connectors, messages, end-points, notifiers, acceptors, streams, schedulers, connection servers, remote object identifiers and dispatchers.

4 Crosscutting Features

Besides offering a synchronous remote method invocation service, AspectJRMI also provides modular implementations of the following crosscutting features: oneway calls, asynchronous calls, call by value-result, service combinators, remote reference decorators and collocation optimization.

4.1 Oneway Calls

In oneway calls, control returns to the client as soon as the middleware layer receives the call. Thus, client and remote method executions are asynchronous. Oneway calls neither return values nor throw remote exceptions. In AspectJRMI, the abstract aspect `OneWayAspect` defines oneway calls. This aspect has the abstract pointcut `onewayCalls` that defines the calls dispatched using oneway semantics.

```
abstract aspect OneWayAspect {
  protected abstract pointcut onewayCalls();
}
```

Example: Aspect `MyOneWayAspect` defines that calls to `void Hello.sayHello()` should be dispatched using oneway semantics.

```
aspect MyOneWayAspect extends OneWayAspect {
  protected pointcut onewayCalls(): call(void Hello.sayHello());
}
```

4.2 Asynchronous Calls

In order to use asynchronous calls, the programmer has to define which methods are intended to be asynchronously called by means of intertype declarations of AspectJ. AspectJRMI provides the aspect `AsynchronousCallsAspect` which encapsulates almost all implementation details of this feature. This aspect relies on pointcut `asynchronousCalls()` to specify asynchronous calls, which are those whose names start with `async_`, return `Future`, and are not performed within `Skeleton` classes.

```
public aspect AsynchronousCallsAspect {
 public pointcut asynchronousCalls():
   call(Future *.async_*(..)) && !(within(*_Skeleton));
}
```

Method `getResult` in class `Future` returns the result of an asynchronous remote call. If this call has not finished, `getResult` waits its conclusion. If this call fails, `getResult` throws an exception of type `RemoteException`.

Example: Consider that the methods in interface `Hello` may be called asynchronously.

```
interface Hello extends Remote {
 String sayHello() throws ArcademisException;
 String sayHello(String name) throws ArcademisException;
}
```

For this purpose, it is only necessary to define an aspect `AsyncHello` that introduces in interface `Hello` an asynchronous version for each of its methods. The asynchronous version must return `Future`, have a prefix `async_` in its name and a body that just returns `null`.

```
aspect AsyncHello {
  public Future Hello.async_sayHello() { return null; }
  public Future Hello.async_sayHello(String name) { return null; }
}
```

The following code fragment asynchronously calls method `async_sayHello` and later uses method `getResult` for synchronizing and getting its result.

```
Future ftr = server.async_sayHello("Bob");
...
String res = (String) ftr.getResult();
```

4.3 Call by Value-Result

Call by value-result is specified by defining formal parameters as having type `Holder`. This class plays the role of a wrapper for the argument passed by value-result. This argument must implement the interface `Marshable`, which denotes serializable values in Arcademis. Class `Holder` has the following interface:

```
class Holder {
  public Holder(Marshable o);
  public Marshable getValue();
  public void setValue(Marshable o);
}
```

On calling a remote method that has a parameter of type `Holder`, a client must: (i) create an instance h of an object of type `Holder` enclosing the actual parameter value; (ii) use h as an actual parameter; and (iii) on return of the method, extract the result from holder h.

In order to use call by value-result, the programmer does not need to define any aspect. The implementation of AspectJRMI uses the following internal aspect to intercept calls having parameters of type `Holder`:

```
pointcut callbyValueResult(): call(* *(..,Holder,..));
```

Example: The following code fragment defines a method with two value-result parameters, as well as a client using this method.

```
void foo(Holder h1, Holder h2) {
  Date d = (Date) h1.getValue();
  d.setDate(12, d.getDay() + 1, 2004);
  h2.setValue(new Date(2, 28, 2005));
}
........
Holder h1 = new Holder(new Date(2,17,2005));
Holder h2 = new Holder(new Date());
foo(h1,h2);
((Date) h1.getValue()).print(); // prints December 18th, 2004
((Date) h2.getValue()).print(); // prints February 28th, 2005
```

In addition, AspectJRMI supports call by result, in which parameters are used to return values from methods. Call by result is specified in a way similar to call by value-result, using type `ResultHolder`.

Restriction: Call by value-result and call by result are incompatible with asynchronous and oneway calls. If this restriction is not followed, a run time error is raised.

4.4 Service Combinators

Service combinators were originally proposed to handle failures and to customize programs that need to retrieve web pages [4]. In AspectJRMI, we adapt this mechanism in order to express that remote invocations should be dispatched sequentially (to provide fault tolerance), concurrently (to decrease response time) or non-deterministically (to provide load distribution).

Let C_1 and C_2 be two remote method calls. AspectJRMI allows the composition of these calls by means of the following service combinators:

- $C_1 > C_2$ (alternative execution): C_1 is invoked; if its invocation fails then C_2 is invoked; if the invocation of C_2 also fails then the combined call fails. Thus, the combinator $>$ provides fault tolerance.
- $C_1 \; ? \; C_2$ (non-deterministic choice): Either C_1 or C_2 is non-deterministically chosen to be invoked. If the selected call fails, then the other one is invoked. The combinator fails when both C_1 and C_2 fail. Thus, service combinator ? provides load balancing.
- $C_1 \mid C_2$ (concurrent execution): Both C_1 and C_2 are concurrently started. The combinator returns the result of the first call that succeeds first; the other one is ignored. The combinator fails when both calls fail. Thus, service combinator | optimizes the response time of an idempotent remote call by concurrently invoking it in two servers.

Service combinators are associated with remote references using the following classes:

```
class SimpleRemoteRef extends RemoteRef {
  public SimpleRemoteRef(Remote endpoint);
}
class StructuredRemoteRef extends RemoteRef {
  public StructuredRemoteRef(char op, RemoteRef r1, RemoteRef r2);
}
```

The abstract class `RemoteRef` represents a remote reference in the system. `SimpleRemoteRef` denotes a standard remote reference, with no service combinator, whereas `StructuredRemoteRef` denotes a remote reference associated with a service combinator. Users can also provide their own subclasses of `RemoteRef` in order to define new combinators.

The abstract aspect `RemoteAspect` associates service combinators with remote references:

```
abstract aspect RemoteAspect {
  protected abstract pointcut RemoteCalls();
  protected abstract RemoteRef getRemoteRef();
}
```

Aspects extending `RemoteAspect` must define the calls with an associated service combinator (pointcut `RemoteCalls`) and the `RemoteRef` used in the invocation of such calls (method `getRemoteRef`).

Example: Suppose the following client of interface `Hello` (Section 4.2).

```
class HelloClient {
  Hello server = RemoteRef.InitRemoteRef();
  String s1 = server.sayHello();
  String s2 = server.sayHello("Bob");
}
```

Suppose we want to use the following tactics in invocations of services of type `Hello`: each call must be first dispatched to the server `helloSrv` in node `skank` ; on failure, it must be dispatched to the server `helloSrv` in node `patofu`. Moreover, this tactic must be associated with calls to `Hello` methods inside class `HelloClient`. The aspect `HelloClientAspect` implements the proposed invocation tactics:

```
 1: aspect HelloClientAspect extends RemoteAspect {
 2:   private RemoteRef ref;
 3:   protected pointcut RemoteCalls(): within(HelloClient)
 4:                                      && call(* Hello.*(..));
 5:   public HelloClientAspect() {
 6:     String s1 = "skank.inf.pucminas.br/helloSrv";
 7:     String s2 = "patofu.inf.pucminas.br/helloSrv";
 8:     ref = new StructuredRemoteRef('>',
 9:                 new SimpleRemoteRef(RmeNaming.lookup(s1)),
10:                 new SimpleRemoteRef(RmeNaming.lookup(s2)));
11:   }
12:   protected RemoteRef getRemoteRef() {
13:     return ref;
14:   }
15: }
```

Line 3 specifies that the invocation tactics supported by this aspect must be associated with calls of methods of type `Hello` occurring inside class `HelloClient`. Lines 8-10 create a structured remote reference using service combinator $>$ for failure recovery.

Restrictions: In order to be properly combined, C_1 and C_2 should represent calls to methods sharing a common name (probably having different target objects). Moreover, the static type checking rules of Java prevents combining of synchronous and asynchronous calls.

4.5 Remote Reference Decorators

Remote reference decorators (also known as interceptors in CORBA) are used to insert additional behavior in the invocation path of remote calls. Decorators use class composition to create a chain of tasks to be executed in the invocation flow of a remote call. A decorator is defined by means of class `RemoteRefDecorator`:

```
abstract class RemoteRefDecorator extends RemoteRef {
  public RemoteRefDecorator(RemoteRef ref);
}
```

The constructor of this class has a parameter `ref` which denotes the remote reference to be decorated. A remote reference decorator should extend the class `RemoteRefDecorator`. AspectJRMI provides the following standard decorators: `Cache` (that implements a cache with the results of idempotent remote calls); `Log` (that provides a log service for remote calls); and `Timer` (that specifies a timeout for the execution of a remote call before throwing an exception).

Example: The following fragment of code associates a `Log` decorator with both components of the structured remote reference of the previous example.

```
 8: ref = new StructuredRemoteRef('>',
 9:          new Log(new SimpleRemoteRef(RmeNaming.lookup(s1))),
10:          new Log(new SimpleRemoteRef(RmeNaming.lookup(s2))));
```

4.6 Collocation Optimizations

In distributed object-oriented systems, there are situations where servers and clients are collocated in the same address space. In such cases, there is no need to dispatch remote calls using the middleware infrastructure. Instead, remote invocations may be directly forwarded to the server object. This strategy is usually called direct collocation optimization[21].

In AspectJRMI, direct collocation does restrict or change the semantics of other concerns associated to an invocations subjected to optimization. Particularly, oneway calls and asynchronous calls can be forwarded to a local object. Moreover, service combinators, remote reference decorators, and call by value-result preserve their semantics in calls subjected to optimizations.

In order to activate collocation optimizations, we need to weave an aspect named `CollocationAspect` with the component application. This aspect has the following interface:

```
aspect CollocationAspect {
  pointcut remoteObjectInit(RemoteObject r):
     initialization(RemoteObject+.new(..)) && this(r);
  pointcut remoteObjectAsResult():
     call(RemoteObject+ *(..));
  pointcut remoteObjectAsParam():
     call(public object Stream.readObject() && within(*_Skeleton);
}
```

Pointcut `remoteObjectInit` captures the initialization of remote objects. An advice associated to this pointcut inserts the identifier of this object in an internal collocation table. Pointcut `remoteObjectAsResult` captures all method

invocations that return a `RemoteObject`. An around advice associated with this aspect checks whether or not the identifier of this remote object is in the collocation table. If it is, a local reference is returned instead of a remote reference. In the lexical scope of skeletons, pointcut `remoteObjectAsParam` captures the deserialization of objects that are passed as parameters in remote calls. Similar to the previous advice, an around advice checks whether or not the deserialized object is local. If it is, a local reference for it is returned.

5 Experimental Results

This section presents results obtained from experiments performed with our implementation of AspectJRMI. The experiments were used to evaluate the size and performance overhead of AspectJRMI.

5.1 Size Overhead

We use the following tools in the experiments: `javac` (JDK 1.4 version), `ajc` (version 1.5.0) and `abc` (version 1.0.2). The `ajc` tool is the default aspect compiler for AspectJ and `abc` is an aspect compiler that incorporates a series of optimizations [1].

Table 1 summarizes the size of the AspectJRMI framework. As reported in Section 3.2, the client version of the core has 37 KB and the client/server version has 68 KB. The internal classes and aspects of AspectJRMI have 50 KB. There are also 41 KB from the AspectJ run-time (package `aspectjrt.jar`). Thus, the total size of the system is 128 KB (client only) and 159 KB (client/server), which we consider a competitive value. For example, full implementations of CORBA, such as the JacORB system [9], have approximately 10 MB. Implementations of CORBA for mobile and embedded devices, such as ORBit2 [16], have at least 2 MB. On the other hand, some implementations of CORBA are smaller than the ones mentioned. For example, the UIC-CORBA system has 48.5 KB in the Windows CE platform and around 100 KB in Windows 2000 [19]. However, UIC-CORBA supports only a basic remote method invocation service.

Component	**Client**	**Server**
Core	37	68
AspectJRMI	50	50
AspectJ	41	41
Total	128	159

Table 1. Size (in KB) of AspectJRMI components

Besides the size of the internal components of the framework, there is also the cost of the weaving process. In order to evaluate such cost, we have implemented the following programs:

– P1: A client that calls, using a oneway semantics, a remote method passing as argument a string with 16 chars and an array with 16 integers.
– P2: A client that calls, using an asynchronous semantics, a remote method that receives as argument two integers and returns a string.
– P3: A client that calls a remote method passing as argument two arrays of integers, with size 128 and 32. The call uses as target a remote reference with an associated `?` service combinator (non-deterministic choice).
– P4: A client that calls a remote method passing as argument a string with 32 chars and two integers. The invocation uses call by value-result.

We have used the `ajc` and `abc` weavers to compile such applications. Furthermore, in order to provide a lower bound for comparison we have changed programs P1 and P2 to use a standard synchronous semantics. Program P3 was changed in order to remove the `?` service combinator. Program P4 was changed to use call by value. The modified and simplified programs were compiled using the standard `javac` compiler.

Table 2 summarizes the results. Column weaving describes the components of the application that were instrumented by the weaver compiler (client or client/server). Columns `ajc` and `abc` show the size of these components after the weaving (using the `ajc` and `abc` compilers). The next column shows the size of these components in the modified programs when compiled using the `javac` compiler. Finally, column `abc-javac` presents the difference in size of the code generated by the two compilers.

	One call	**Weaving**	**ajc**	**abc**	**javac**	**abc-javac**
P1	Oneway	Client	4.99	2.46	0.84	1.62
P2	Asynchronous	Client	11.00	5.76	2.84	2.92
P3	Combinator ?	Client	5.76	3.21	0.81	2.40
P4	Value-result	Client/Server	8.67	5.53	1.92	3.61

Table 2. Size (in KB) of the experiments (for one call)

The results of the first experiment show that the `ajc` compiler introduces a considerable size overhead. Certainly, the reason is that this compiler does not support many optimizations that are possible in aspect-oriented languages [2]. On the other hand, the results using the `abc` are much more acceptable. When compared to the `javac` programs, the overhead ranges from 1.62 KB (for program P1) to 3.61 KB (for program P4). This overhead is for one remote call that adds a new feature to both programs (oneway semantics in the case of program P1 and a service combinator in program P3).

In the second group of experiments, we have changed programs P1 to P4 to make 100 remote calls sequentially (one call after the other, without the use of loops). Table 3 presents the results. Considering only the `abc` compiler, the overhead was significantly reduced, ranging from 30 bytes per remote call

(programs P2 and P4) to 100 bytes per remote call (program P3). We believe this overhead is fully acceptable.

	100 calls	**Weaving**	**ajc**	**abc**	**javac**	**(abc-javac)/100**
P1	Oneway	Client	47.9	11.5	2.07	0.09
P2	Asynchronous	Client	11.0	5.76	2.84	0.03
P3	Combinator ?	Client	48.9	12.2	2.05	0.10
P4	Value-Result	Client/Server	8.67	5.53	1.92	0.03

Table 3. Size (in KB) of the experiments (for 100 calls)

5.2 Performance Overhead

Oneway Calls, Asynchronous Calls and Call by Value-Result: In order to evaluate the performance of such features we have reused programs P1, P2, and P4 from the previous section. We run such programs (client and server processes) on a Pentium 4 machine, with 2.00 GHZ, 512 KB RAM, Microsoft Windows Service Pack 4, JDK 5.0 and `abc` aspect compiler (version 1.0.2). Each remote invocation was executed 5000 times. In program P2, we measured the time to dispatch the asynchronous calls, store the `Future` values in an array and retrieve all results from this array (using the `getResult` method). To establish a comparison, we ported programs P1, P2, and P4 to JacORB, preserving their semantics. Program P3 was excluded from the experiment since in JacORB there is no support to service combinators. Table 4 presents the results in calls/second. As showed in this table, the performance of AspectJRMI is very close to JacORB performance.

		AspectJRMI (A)	**JacORB (B)**	**A / B**
P1	Oneway	3595	3426	1.04
P2	Asynchronous	2028	2159	0.94
P4	Value-Result	2192	2153	1.02

Table 4. Throughput (in calls/sec)

Collocation Optimizations: A second experiment was conducted in order to measure the performance gains of collocation optimizations. The following program was used in this experiment:

```
1: s.f1(b1);        // remote call (b1 is a local object)
2: B b2= s.f2()     // remote call that returns a reference to b1
3: b2.g();          // optimization: b2.g() ==> b1.g()
```

The same program has been compiled and executed with and without optimizations. We execute each remote call (lines 1 to 3) 5000 times, with client and server processes in the same machine. All methods have an empty body. Table 5 presents the results in calls/msec. When the collocation aspect was woven to the application we observed a small reduction in the throughput of remote calls not subjected to optimization (lines 1 and 2). This was due to the need to check if the arguments or the results are local objects. On the other hand, calls subjected to optimization (line 3) achieved a substantial performance gain. This result was expected since all the middleware overhead was fully eliminated, including marshalling, unmarshalling, and TCP/IP communication. Improvements in the same order of magnitude were already reported for direct collocation optimization in CORBA [13, 21].

Line	Call	Disabled (A)	Enabled (B)	B/A
1	s.f1(b)	3.36	3.31	0.98
2	s.f2()	3.51	3.50	0.99
3	b2.g()	3.40	833.33	245.09

Table 5. Calls/msec with collocation optimization enabled and disabled

6 Related Work

Customizable and Adaptive Middleware: Several software engineering techniques have been applied in the construction of customizable, open and adaptive middleware platforms. Computational reflection, for example, is the central concept of systems such as openORB [3], openCOM [5], UIC [19], and dynamicTAO [12]. However, reflective middleware systems often provide low level APIs and introduce non-marginal performance and memory overheads. Systems such as Quarterware [22] and Arcademis [17] rely on the concept of frameworks. These systems provide semi-complete platforms that can be extended and personalized by middleware users. Other systems, such as TAO [20], relies on design patterns. However, frameworks and design patterns do not provide modularized implementation for crosscutting features, such as interceptors, oneway calls, asynchronous calls, value-result parameter passing, and collocation optimizations. Therefore, it is usually difficult to remove or add such features in the middleware platform. Particularly, even if a crosscutting feature is not requested in a particular application, the middleware carries code to support its implementation. This increments the size of the system and may impact its performance.

AOP Refactorization of ORBacus: Using aspect mining techniques, Zhang and Jacobsen have quantified the crosscutting nature of several features of CORBA based middleware [26, 27]. They have measured the scattering degree of features such as portable interceptors, dynamic programming invocations, collocation

optimizations, and asynchronous calls. They have also showed that such features can be modularized using aspect-oriented programming. From this experience, they proposed the horizontal decomposition method [28]. They have assessed the effectiveness of their method by re-implementing as aspects crosscutting features of the original implementation of ORBacus [15]. As expected, their refactorization preserves the CORBA programming interface.

AspectJRMI design was not constrained by a predefined programming interface. Similar to Java RMI [24], AspectJRMI does not extend the Java language nor require a particular interface definition language. Instead, the system leverages the pointcut mechanism of AspectJ to define features such as oneway calls, service combinators, and decorators. Intertype declarations are used to define asynchronous calls. This contributes to increase the degree of obliviousness provided by the middleware API.

Just-in-time Middleware and Abacus: Following horizontal decomposition guidelines, the Just-in-time Middleware (JiM) paradigm [25] advocates that middleware implementation should be pos-postulated, i.e., middleware components should be selected and assembled after the user application is specified. Abacus is a prototype implementation of JiM principles. The system relies on an aspect-aware compiler, called Arachne, to generate just-in-time middleware configurations. Arachne collects middleware functionalities from IDL declarations and from a user interface. The synthesis process also depends on dependencies, constraints and convolution descriptions. Dependencies define that the implementation of a certain feature is composed by other ones. Constraints specify that some features must be included or excluded from the system depending on external conditions. Convolutions descriptions specify that a feature crosscuts the implementation of other features. On the other hand, AspectJRMI shows that a standard AOP language and weaver can be used to associate pos-postulated features to a minimal core middleware system. From the list of aspects available in Abacus, AspectJRMI does not provide support only to server data types (since the system relies on the Java type system) and some CORBA advanced features (such as interface repository and dynamic invocation interface). However, we believe that AspectJRMI can be extended to include such missing aspects.

Other Applications of AOP to Middleware: JBossAOP [10] uses Java 1.5 annotations to support the implementation of several concerns, including oneway calls, asynchronous calls, transactions and persistence. Basically, in JBossAOP annotations provide syntactical hooks to insert aspectual code. However, one can argue that in this case annotations introduce code scattering, since they can be required in several elements of the system. Preferably, annotations should be used to express functional behavior of the annotated element, for example, to define that an operation is idempotent.

Alice [8] is a middleware that proposes the combination of aspects and annotations to implement container services, such as authentication and sessions.

AspectJ2EE [6] proposes the use of AOP to implement open, flexible and extensible container middleware. Soares, Laureano and Borba [23] have proposed a set of guidelines to implement distribution, persistence and transactions as aspects. The previous systems, however, consider the underlying communication middleware as a monolithic block.

FACET [18] is an implementation of a CORBA event channel that relies on AOP to provide a customizable event system. Similar to AspectJRMI, the system has a core and a set of selectable features. Each feature adds a new functionality to the core or to other feature. Examples of features include pulling events, dispatching strategies, event payload types, event correlation and filtering, and event profile. FACET also includes a test framework that automatically validates combinations of features. Thus, FACET design follows many of the horizontal decomposition principles.

7 Conclusion

In this paper we described an aspect-oriented communication middleware system with high degree of modularization, configurability and customizability. The design of the system has followed the horizontal decomposition principles, and thus AspectJRMI has a set of mandatory components (middleware core) designed using traditional vertical decomposition techniques. Furthermore, AspectJRMI uses aspects to encapsulate optional features so that users can select only those features that are needed in a particular distributed application. An aspect compiler is used to weave aspects and mandatory components at compile time.

The core of the system is derived from components defined in Arcademis, a Java-based framework that supports the implementation of customizable middleware architectures. Well-known design patterns, such as singletons, factories, strategies, decorators and façades, are used to foster a non-monolithic and flexible middleware core. Users can change and extend almost all internal components of the core, including channels, invokers, service handlers, streams, remote references, dispatchers, and activators. Despite its open architecture, the core of AspectJRMI has just 37 KB (client features only) or 68 KB (client and server features). For example, we were able to successfully run the core in the CLDC configuration of the J2ME platform. This configuration targets resource constrained devices, such as cell phones and low-end PDAs.

The core provides a primitive remote invocation service (synchronous, using call by value and at-most-once semantics). Whenever crosscutting functionalities are required, they can be introduced using aspects. AspectJRMI provides aspects for the following features: oneway calls, asynchronous calls, service combinators, remote reference decorators, value-result and result parameter passing, and direct collocation optimizations. All these aspects when packed have around 50 KB. When using the `abc` AspectJ compiler the cost of weaving aspects to the base application ranges from 30 to 100 bytes per remote call. This makes AspectJRMI a competitive solution when compared to other middleware implementations. For example, minimal CORBA implementations that are equivalent

to AspectJRMI core have around 50 to 100 KB [19]. On the other hand, full implementations of CORBA have at least 2 MB [16]. Thus, AspectJRMI shows that aspect-oriented programming is an effective solution to provide middleware systems with size and functionalities between these bounds. Our experiments have also shown that our static weaving approach has not significantly impacted middleware performance. We were able to obtain performance results equivalent to mature CORBA implementations.

Differently from CORBA, AspectJRMI is a solution to Java-based distributed applications. For this reason, the proposed middleware does not require specific interface definition languages and type systems. The middleware also does not require extensions to the Java object model or execution environment. The same approach is followed by Java RMI, which provides just a basic invocation service, supported by a monolithic kernel. In AspectJRMI, aspect-oriented abstractions, such as pointcuts and intertype declarations, are used to add extra features to an open and non-monolithic kernel. The only requirement is that users should be familiar with an aspect-oriented language (AspectJ, in the case).

As future work, we intend to support other non-functional requirements, besides distribution. We plan to investigate support to aspects such as security, load balancing, fault tolerance and persistence.

References

1. P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.
2. P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128. ACM Press, 2005.
3. G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, H. A. Duran-Limon, T. Fitzpatrick, L. Johnston, R. S. Moreira, N. Parlavantzas, and K. B. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.
4. L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, 1999.
5. M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware: IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 160–178. Springer-Verlag, 2001.
6. T. Cohen and J. Gil. AspectJ2EE = AOP + J2EE. In *18th European Conference Object-Oriented Programming*, volume 3086 of *LNCS*, pages 219–243. Springer-Verlag, 2004.
7. A. Colyer and A. Clement. Large-scale AOSD for middleware. In *3rd International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

8. M. Eichberg and M. Mezini. Alice: Modularization of middleware using aspect-oriented programming. In *4th International Workshop on Software Engineering and Middleware*, volume 3437 of *LNCS*, pages 47–63. Springer-Verlag, 2005.
9. JacORB. http://www.jacorb.org.
10. JBossAOP. http://www.jboss.org/developers/projects/jboss/aop.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–355. Springer Verlag, 2001.
12. F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhes, and R. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms*, volume 1795 of *LNCS*, pages 121–143. Springer-Verlag, 2000.
13. A. S. Krishna, D. C. Schmidt, K. Raman, and R. Klefstad. Enhancing real-time CORBA predictability and performance. In *International Symposium on Distributed Objects and Applications*, volume 2888 of *LNCS*, pages 1092–1109, 2003.
14. Object Management Group. The common object request broker: Architecture and specification revision 3.0.2, Dec. 2002.
15. Orbacus. http://www.orbacus.com.
16. ORBit2. http://orbit-resource.sourceforge.net.
17. F. M. Pereira, M. T. Valente, R. Bigonha, and M. Bigonha. Arcademis: A framework for object oriented communication middleware development. *Software Practice and Experience*, 2005. to appear.
18. R. Pratap. Efficient customizable middleware. Master's thesis, Department of Computer Science and Engineering, Washington University, 2003.
19. M. Román, F. Kon, and R. Campbell. Reflective middleware: From your desk to your hand. *Distributed Systems Online*, 2(5), July 2001.
20. D. C. Schmidt and C. Cleeland. Applying patterns to develop extensible and maintainable ORB middleware. *IEEE Communications*, 37(4):54 – 63, 1999.
21. D. C. Schmidt, N. Wang, and S. Vinoski. Object interconnections collocation optimizations for CORBA. *SIGS C++ Report*, 10(9), 1999.
22. A. Singhai, A. Sane, and R. H. Campbell. Quarterware for middleware. In *18th International Conference on Distributed Computing Systems (ICDCS)*, pages 192–201. IEEE Computer Society, 1998.
23. S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *17th ACM Conference on Object-Oriented programming systems, languages, and applications*, pages 174–190. ACM Press, 2002.
24. A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX, 1996.
25. C. Zhang, D. Gao, and H.-A. Jacobsen. Towards just-in-time middleware architectures. In *4th International Conference on Aspect-Oriented Software Development*, pages 63–74. ACM Press, 2005.
26. C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *2nd International Conference on Aspect-Oriented Software Development*, pages 130–139. ACM Press, 2003.
27. C. Zhang and H.-A. Jacobsen. Refactoring middleware with aspects. *IEEE Transactions Parallel and Distributed Systems*, 14(11):1058–1073, 2003.
28. C. Zhang and H.-A. Jacobsen. Resolving feature convolution in middleware systems. In *19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 188–205. ACM Press, 2004.