Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1659–1666
The Journal of Systems and Software
www.elsevier.com/locate/jss

Collocation optimizations in an aspect-oriented middleware system

Marco Tulio Valente *, Rodrigo Palhares

Department of Computer Science, PUC Minas, Av. Dom Jose Gaspar, 500 CEP: 30535-610 Belo Horizonte, MG, Brazil

Received 3 July 2006; received in revised form 25 December 2006; accepted 3 January 2007
Available online 7 February 2007

Abstract

In distributed object-oriented systems, there are situations where client and server objects are deployed in the same address space. In such scenarios, it is possible to dispatch remote calls without having to transverse the infrastructure provided by the underlying communication middleware system and thus without incurring the overhead of using a networking loopback interface. Such optimizations are called collocation optimizations. In this paper we describe an implementation of collocation that is centered on aspect-oriented programming abstractions. This implementation provides high degrees of modularization, configurability and adaptability than current object-oriented support to collocation. The paper also presents results about the performance gains derived from the optimization proposed.

Keywords: Middleware; Collocation optimization; Aspect-oriented programming

1. Introduction

Remote method invocation is the key abstraction provided by communication middleware systems, such as CORBA (Object Management Group, 2002) and Java RMI (Wollrath et al., 1996). Using this abstraction, client processes can call methods of remote objects using the same syntax of local calls. In order to provide transparency in remote communication, middleware systems encapsulate several details inherent to distributed programming, including communication protocols, data marshalling and unmarshalling, heterogeneity, service lookup, synchronization, and failure handling. However, in distributed settings there are situations where client and server objects are deployed in the same address space or situations where one of the objects migrate to the address space of the other. In such scenarios, it is possible to dispatch remote calls without having to transverse the middleware infrastructure and thus without incurring the overhead of using a networking loopback interface (as showed in Fig. 1). In communication middleware systems, such optimizations are called *collocation optimizations* and are usually supported by middleware platforms such as TAO (Schmidt and Cleeland, 1999), ORBacus (Orbacus), ICE (Henning, 2004) and COM/DCOM (Box, 1997).

Implementations of collocation usually present a crosscutting behavior (Zhang and Jacobsen, 2004). Each time the middleware layer obtains a remote reference,[1] it has to check if this reference is associated to a local object, i.e., an object that resides in the same address space of the object that has requested the reference. This verification requires modifications at least in the following components of a middleware system: stubs (in order to check if the result of a remote call is a local object), skeletons (in order to check if the arguments of a remote call are local objects) and in the kernel of the system (in order to maintain a table with references to local objects. This table is used to support the tests that stubs and skeletons perform to determine

* Corresponding author. Tel./fax: +55 31 3439 5204.
E-mail addresses: mtov@pucminas.br (M.T. Valente), palhares.rodrigo@gmail.com (R. Palhares).

[1] A remote reference is an abstraction that denotes a reference for an object that can accept remote method calls. From an implementation point of view, a remote reference points to a middleware internal component called stub. 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.

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.01.033

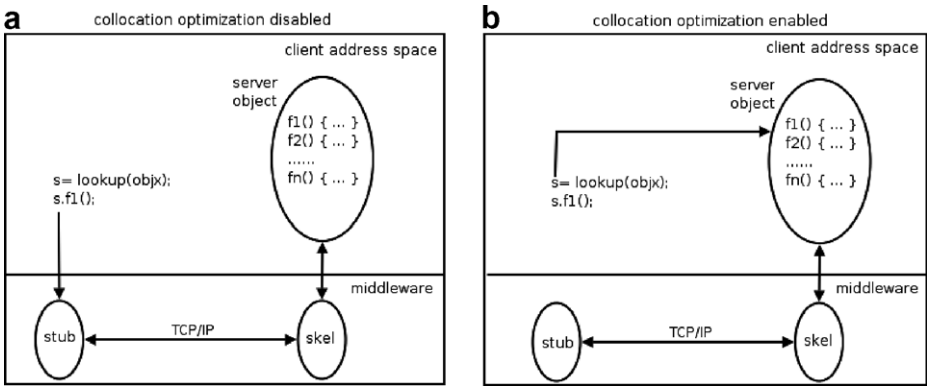

Fig. 1. Remote calls when client and server objects are in the same address space: (a) without collocation optimization and (b) with collocation optimization enabled.

collocation). Moreover, collocation optimizations usually require the extension of middleware systems with new components, such as specialized stubs, that ensure that calls subjected to optimizations are dispatched using the standard semantics employed by middleware systems. For example, specialized stubs can be required to preserve the semantics of parameter passing or to support middleware specific features such as interceptors, oneway calls and thread policies.

For this reason, the implementation of collocation optimizations in conventional middleware systems do not present expected levels of modularization and separation of concerns, which makes such systems more difficult to implement, maintain and evolve. Moreover, middleware platforms with support to collocation always carry code to implement this feature, even when it is known in advance that collocation tests are not helpful in a given distributed application. For example, usually collocation is not effective in real time systems, where object deployment is carefully planned by application developers. In such situations, middleware users do not have the chance to rebuild the kernel of the system in order to remove the code used to determine collocation and as result configure a system with small memory footprint requirements.

In this paper, we describe an implementation of collocation optimization that is centered on aspect-oriented programming abstractions and that was conducted over the communication infrastructure provided by AspectJRMI (Valente et al., 2005). AspectJRMI is a middleware system structured over a kernel that provides a primitive remote invocation service (synchronous, using call by-value and at-most-once semantics). Whenever services that crosscut the kernel functionality are required, they can be introduced using aspects. The current version of AspectJRMI provides aspects for the following crosscutting features: oneway calls, asynchronous calls, service combinators, remote reference decorators, and value-result parameter passing. AspectJRMI users can configure an aspect compiler to just combine the middleware components (or aspects) that are effectively needed in the distributed application they are building.

The remainder of this paper describes aspects that incorporate collocation optimizations in the AspectJRMI kernel. Section 2 presents the design principles and the main services provided by AspectJRMI. Section 3 describes the programming interface and the aspects proposed to handle collocation in AspectJRMI. Section 4 describes an alternative type of collocation that preserves the original semantics of parameter passing in remote calls subjected to optimizations. Section 5 describes an experimental evaluation of the performance gains derived from collocation. Section 6 describes related work and Section 7 concludes the paper.

2. AspectJRMI

AspectJRMI (Valente et al., 2005) is a Java-based communication middleware that applies aspect-oriented programming concepts to achieve high degrees of configurability and pluggability. The design of AspectJRMI has followed a set of principles, called horizontal decomposition (Zhang and Jacobsen, 2004), that advocates the synergistic combination of classes and aspects in order to modularize middleware concerns. The components of AspectJRMI are divided in two categories: mandatory and non-mandatory.

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 factories that create channels that use different transport protocols or they can define decorators that add extra functions to channels (such as encryption or compression of messages). Mandatory components were implemented using traditional object-oriented programming techniques (such as design patterns and vertical decomposition). Particularly, the core of the system was implemented using components defined

in Arcademis (Pereira et al., 2006), a Java-based framework to support the implementation of middleware systems.

Non-mandatory components implement features that are not required in every distributed application. Most of these features have a crosscutting behavior (Zhang and Jacobsen, 2003) and thus they are implemented as aspects using AspectJ. AspectJRMI provides modular and pluggable implementations for the following features:

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.

Asynchronous calls. Similar to oneway calls, asynchronous calls are handled by the middleware in a different thread. However, asynchronous calls return a `Future` object, that can be used to poll for the result of the call.

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.

Service combinators. Service combinators were originally proposed to handle failures and to customize programs that need to retrieve web pages (Cardelli and Davies, 1999). 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 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 $\mid$ optimizes the response time of a remote call by concurrently invoking it in two servers.

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 take advantage of class composition to create a chain of tasks that are executed in the invocation flow of a remote call.

3. Collocation optimizations in AspectJRMI

In this section, we describe the programming interface and the aspects that handle collocation optimizations in AspectJRMI. First, Section 3.1 presents the middleware programming interface that supports collocation. Next, Section 3.2 presents the aspects that weave such optimizations in the kernel of AspectJRMI.

3.1. Programming interface

As described in Section 2, AspectJRMI relies on components provided by a framework called Arcademis (Pereira et al., 2006). Particularly, remote methods must be declared in an interface that extends the `Remote` interface and must declare the possibility of throwing `ArcademisException`. Remote object classes must implement this interface and extend the `ArcademisRemoteObject` class. The following code shows the creation, activation and registration of an object whose class implements the remote interface `A`:

```
1: interface A extends Remote {
2:   public int foo(int a,float b) throws ArcademisException;
3: }
4: class A_Impl extends ArcademisRemoteObject implements A {
5:   ...
6:   public int foo(int a,float b) throws ArcademisException {
7:     ...
8:   }
9:   public static void main(String[] args) {
10:    ...
11:    A a=new A_Impl();
12:    ArcademisNaming.bind(``objx",a);
13:    a.activate();
14:    ...
15:  }
16: }
```

The following code fragment queries the AspectJRMI naming service for a remote reference to an object associated to the name `bar`. Next, a remote call is performed using this reference.

```
l7: A a= (A) ArcademisNaming.lookup(‘‘bar");
l8: int x= a.foo(l0,l.l);
```

If the client object that has dispatched the remote call (line 18) and the server object that handles it (and that was created in line 11) are located in the same address space, the middleware layer will transparently transform such call in a local call, i.e., a call that bypasses the middleware stack and the TCP/IP loopback interface.

3.1.1. Thread semantics

Synchronous remote calls subjected to collocation optimization are dispatched in the client thread instead of being dispatched using a separate thread. On the other hand, collocation preserves the dispatching semantics of oneway and asynchronous calls in different threads. It is worth to mention that AspectJRMI servers use a thread per connection semantics when dispatching remote calls. Thus, despite if collocation is applied or not, remote object implementations must be thread safe.

3.1.2. Enabling and disabling collocation

By default, collocation optimizations are enabled in any distributed application built using AspectJRMI. However, at compilation time, developers can decide to disable collocation verification in their applications – for example, to reduce memory footprint requirements. For this purpose, they just need to remove the collocation aspects (described in Section 3.2) from the weaving process. Moreover, at runtime, developers can decide to disable (or enable) collocation for a given remote object, by calling the method `collocation` introduced by AspectJRMI collocation aspects in the class `ArcademisRemoteObject`:

```
A a = new A_Impl( );
a. collocation(false);
```

Developers can also decide to disable collocation for all remote objects by calling the static method `collocation` available in the aspect `CollocationAspect`:

```
CollocationAspect.aspectOf( ).collocation
(false);
```

A global collocation configuration overrides local collocation policies defined for any remote object.

3.2. Implementation

The implementation of collocation optimization in Aspect-JRMI relies on aspects to perform the following tasks:

Intercept the creation of objects that can be affected by collocation. In order to accomplish this task, the following pointcut is defined:

```
pointcut newCollocatedObject(ArcademisRe-
moteObject r):
  execution(Remote + .new(..)) && this(r);
```

This pointcut captures the execution of the constructors of classes that implement the interface `Remote`, i.e., objects the middleware must monitor to determine if they are eventually collocated with their clients. An after advice associated to this pointcut inserts a reference to the created object in a hash table, called collocation table. The key of the entry inserted in this table is a global an unique identifier for the created object. Moreover, references stored in the collocation table are weak, i.e., they are not considered for the purpose of garbage collection.

Accordingly to the Absolute Object Reference remoting pattern (Volter et al., 2005), stubs in AspectJRMI have an attributed of type `RemoteReference` that contains information about the address of the remote server, including its host name and port number. Each remote object has a dedicated server request handler that encapsulates a thread that continuously accepts connections in the port saved in the remote reference associated to this server object. The server request handler directly forwards incoming connections to the skeleton, that unmarshals the arguments from the marshal stream and invokes the remote method. Thus, the server side of AspectJRMI applications does not maintain a table that maps remote object identifiers to remote object implementations.[2] For this reason, the pointcut described earlier is responsible for creating and keeping such table updated.

Intercept the retrieval of remote references. In object-oriented middleware systems, there are two ways an application can obtain a reference to an object that resides in a remote address space: as a result of a remote call (including calls to the naming service) and as an argument of a remote call. Thus, the implementation of collocation optimization in AspectJRMI must: (1) intercept the retrieval of remote references; (2) extract the global and unique identifier of the object denoted by this reference; (3) lookup for this identifier in the collocation table of the current process; (4) if the lookup is successful, an around advice must return to the calling process the reference stored in the collocation table, instead of the original remote reference. As described, collocation tables maintain standard Java references that directly denotes local server objects.

In order to intercept methods that return remote references as result, the following pointcut is employed:

```
pointcut RemoteReferenceAsResult( ):
  call(Remote+ *(..));
```

[2] This is a AspectJRMI design decision instead of a limitation imposed by Arcademis. Particularly, it is possible to derive middleware systems from Arcademis where a dispatcher component manages connections for several remote objects. In such middleware systems, the dispatcher should keep a table that maps object identifiers to object references. Thus, this table would preclude the maintenance of a separate collocation table.

Moreover, in order to intercept methods that receive remote references as arguments, the following pointcut is used:

```
pointcut RemoteReferenceAsArgument( ):
  call(public Object Stream.readObject( ))
& & within(*_Skeleton).
```

This pointcut intercepts calls to the method `readObject` performed in the lexical scope of classes that represent skeletons in AspectJRMI. In such classes, the `readObject` method is called to unmarshal objects. An around advice associated to this pointcut is first used to obtain a reference to the unmarshalled object. If this reference denotes a local object (i.e., an object accessible from the collocated table), the advice returns a reference to the local object instead of the original reference returned by the `readObject` invocation.

4. Indirect collocation

The implementation of collocation described earlier in this paper is classified as direct collocation (Schmidt et al., 1999), since after optimization is applied clients forward all requests directly to the server object. In other words, direct collocation replaces a remote reference (that denotes a stub in the middleware layer) by a standard Java reference. For this reason, direct collocation ensures that calls subjected to optimizations have exactly the same performance of local calls.

However, there are situations where remote calls have a semantics different from local calls. For example, in AspectJRMI objects that implement the `Marshalable` interface are passed by value when used as arguments of remote calls. However, if such calls are optimized using direct collocation, the called method will receive as argument a built-in Java reference for the marshalable object (instead of a copy of such argument generated during the marshalling process by the middleware layer when no optimization is applied).

For this reason, AspectJRMI provides support for a second type of collocation, called indirect collocation. This type of collocation preserves the original semantics of parameter passing in remote calls subjected to optimizations. In order to enable (or disable) indirect collocation for a given remote object, developers must use the method `indirectCollocation` introduced by AspectJRMI in the class `ArcademisRemoteObject`:

```
A a = new A_Impl( );
a. indirectCollocation(true);
```

AspectJRMI developers can also decide to enable indirect collocation for all remote objects by calling the static method `indirectCollocation` available in the `CollocationAspect` aspect:

```
CollocationAspect.aspectOf( ).indirect
Collocation(true);
```

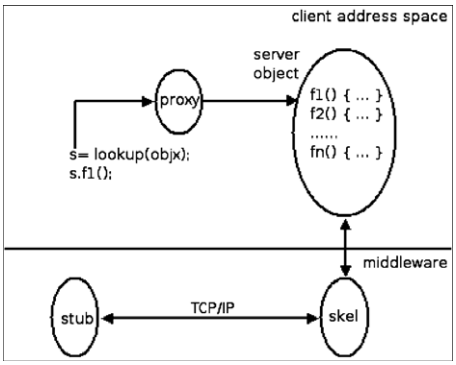

Fig. 2. Indirect collocation optimization.

In order to support indirect collocation, pointcuts similar to the ones described in Section 3.2 are provided. However, the advices associated to such pointcuts return a proxy to the server object (as described in Fig. 2). This proxy is created dynamically, using the dynamic proxy class feature of the Java reflection API. These classes have two distinguished properties: (i) their bytecode is manufactured at runtime by the Java reflection API; (ii) they implement a list of interfaces specified at creation time. Instances of a dynamic proxy classes have an associated invocation handler object, informed by the client that requested its creation. Any method invocation on a dynamic proxy class instance is automatically dispatched to the `invoke` method of the associated invocation handler.

In case of proxies created to handle indirect collocation, the invocation handler checks the list of arguments of the remote call for objects that implement the `Marshalable` interface. If an argument implements this interface, it must be passed using call by-value. For this purpose, the invocation handler creates a clone for such argument and stores a reference to the clone in the list of arguments of the call (replacing the reference to the original argument). Next, the invocation handler forwards the call to the server object.

5. Experimental results

This section presents results obtained from experiments performed with our AspectJRMI implementation of collocation optimization. The experiments were used to evaluate the gains obtained with such optimizations. We have run the experiments in a 3.0 GHz Pentium 4 machine, with 1 GB RAM, JDK 1.5.0 and AspectJ 1.1. Two operating systems were used: Microsoft Windows XP Service Pack 2 and Fedora Core 4 (kernel 2.6.11). In all the experiments, client and server objects were deployed in the same JVM. When collocation optimizations are not enable, the measured times include the overhead of using the communication infrastructure provided by AspectJRMI, including marshalling and unmarshalling functions, middleware protocol, TCP/IP loopback facility, etc.

The first experiment makes use of a remote server object that exports the following method:

```
void foo (long n) throws ArcademisException {
  double a = Math.random( ), b = Math.random( );
  for(long i = 0; i < n; i++) {
    a=a*(b*(a*(b*(a*(b*(a*(a*(b*a)))))))));
    a = Math.random( ); b = Math.random( );
  }
}
```

From a client object instantiated in the same JVM of the remote server, the method `foo` was invoked several times, changing the parameter `n` that represents the number of iterations. In the first sequence of invocations, collocation optimizations were disabled, i.e., aspects supporting this feature were not combined with the core distributed application. In the second sequence of invocations, we activated direct collocation optimizations. Fig. 3 presents the number of invocations by milliseconds in both cases.

As presented in Fig. 3, the performance gains derived from weaving the collocation optimization aspects are considerable. In the Windows operating system, for two hundred iterations, we have almost three times more calls/ms when using optimizations (11.42 against 4.00 calls/ms). However, this difference decreases as the number of iterations increases. For two thousands iterations, we have almost the same number of calls/ms (1.18 against 0.97 calls/ms). This behavior is expected in any collocation implementation, since for long running methods the overhead generated by the middleware is insignificant when compared with the execution time of the method (Pilhofer, 1999). When using Linux, the performance gains were similar. In fact, the number of collocated calls/ms was almost equal to the numbers obtained in Windows. For this reason, such values are not plotted in Fig. 3. On the other hand, when collocation was disabled, the performance using Linux was superior to Windows, predominantly for a small number of iterations. For example, for two hundred iterations, Linux has achieved 4.90 calls/ms against 4.00 calls/ms of Windows. For two thousand iterations, the throughput was almost identical on both operating systems (1.03 calls/ms on Linux against 0.97 calls/ms on Windows).

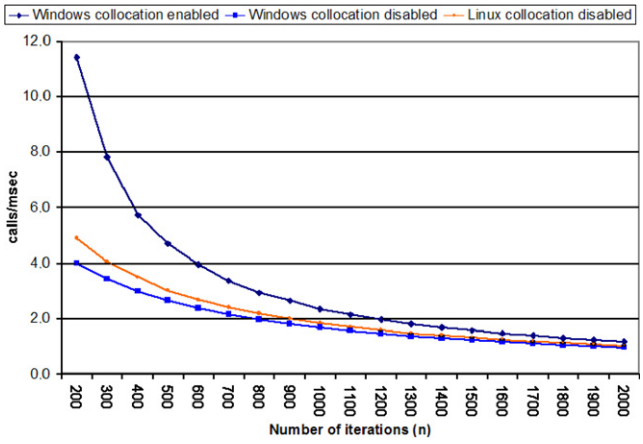

Fig. 3. Direct collocation performance.

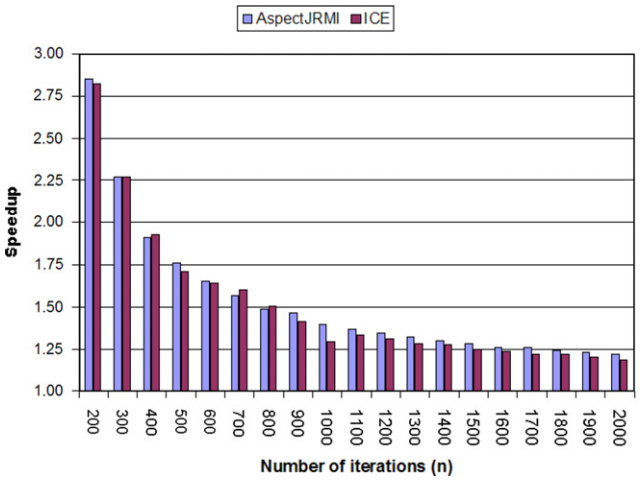

Fig. 4. AspectJRMI and ICE collocation speedup.

In order to compare AspectJRMI collocation performance with other middleware systems, we have implemented the same experiment using the Java implementation of ICE (Henning, 2004). Fig. 4 presents the speedup generated by collocation in the Windows version of both systems (i.e., the figure shows the ratio of calls/ms when direct collocation is enabled to calls/ms when the optimization is disabled). AspectJRMI speedup was slightly superior to ICE in most of the iterations measured. Since ICE is well-known for its performance and scalability properties – particularly when compared with CORBA systems – we consider very acceptable the performance increment generated by collocation in AspectJRMI.

The second experiment was designed in order to evaluate the performance gains derived from using indirect collocation optimization (on the Windows operating system). The experiment relies on the following remote method:

```
1: public Node bar(Node v) throws Arcademis-
Exception {
2: return v;
3: }
```

The class `Node` contains the following attributes: an integer, a float, a double and strings with 16, 32, 64 and 128 characters. The class also contains a reference to a next `Node`, in order to implement a list. For measuring the performance gains offered by indirect collocation, several remote calls where requested, changing the size of the list passed as argument and returned by the `bar` method. As Fig. 5 shows, the gains of using indirect collocation are significant. As expected, increasing the size of the list, the performance of both calls (with indirect collocation enabled and with collocation disabled) decrease. However, optimized calls continue with a better performance, as can be figured out in Fig. 6 that presents the speedup generated by indirect collocation. In this experiment, we have not compared the gains generated by direct collocation, since in this form of collocation the overhead of marshalling and unmarshalling the list does not exist.

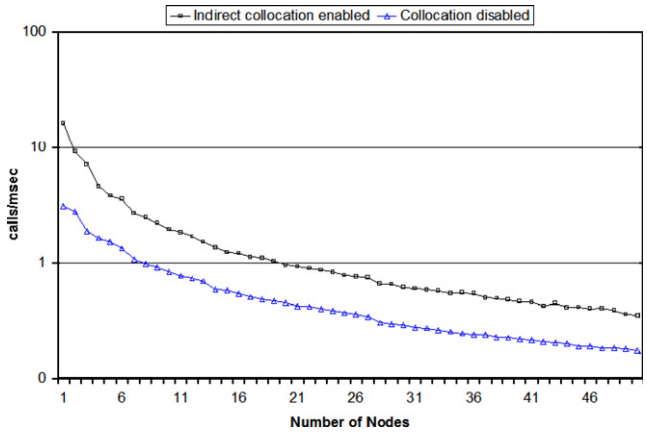

Fig. 5. Indirect collocation performance.

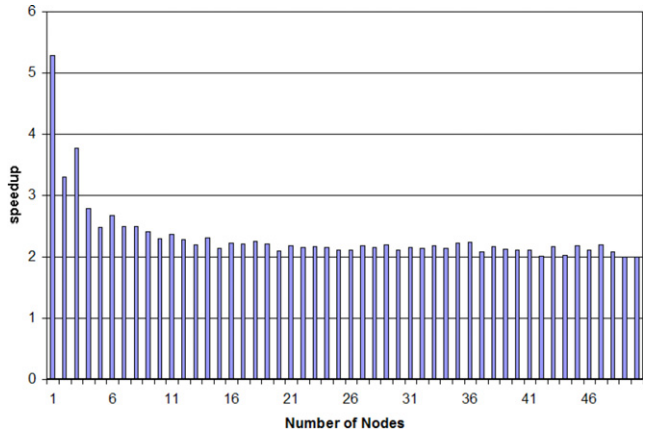

Fig. 6. Indirect collocation speedup.

6. Related work

Several middleware platforms provide support to collocation optimizations, including early systems such as the Spring object-oriented operating system (Radia et al., 1995), proposed in the late eighties. More recently, collocation has been incorporated in many CORBA implementations. TAO (Schmidt and Cleeland, 1999) is a high-performance and real-time CORBA system whose implementation has been guided by a set of optimization principle patterns (Pyarali et al., 1999). Essentially, such patterns are applied in TAO to increase the performance, scalability and predictability of distributed systems built using the middleware. TAO supports both direct and indirect collocation. Benchmarks performed with the C++ implementation of the system demonstrate that the gains achieved with collocation are also significant in other programming languages (Schmidt et al., 1999; Pyarali et al., 1999).

MICO (Puder and Romer, 2000), ORBit (ORBit2) and ORBacus (Orbacus) are other CORBA based platforms that incorporate collocation optimizations. However, the implementation of collocation in such systems is contaminated by a well-known problem of mainstream CORBA implementations: the lack of flexibility, configurability, adaptability and customizability (Zhang and Jacobsen, 2004). Tangled and invasive implementations of collocation have also been incorporated in other middleware systems, including ICE (Henning, 2004), COM/DCOM (Box, 1997) and CORBA Component Model (Siegel, 2000).

Using aspect mining techniques, Zhang and Jacobsen have quantified the crosscutting nature of several CORBA services, including collocation and other features such as portable interceptors, dynamic programming invocations, and asynchronous calls (Zhang and Jacobsen, 2003). They have also showed that such features can be modularized using aspect-oriented programming. From their experience, they have proposed the horizontal decomposition method (Zhang and Jacobsen, 2004), which has been followed in the design of AspectJRMI.

Besides collocation, there are attempts to apply aspect-oriented techniques in the implement of other middleware features. Putrycz and Bernard describe the use of aspects 'incorporate a load balancing service to ORBacus (Putrycz and Bernard, 2002). Aspects are used to detect server replicas and to capture remote calls and if necessary redirect them to the replica suggested by the system load balancing strategies. FACET (Pratap, 2003) is an aspect-oriented implementation of a event system. Similar to AspectJRMI, the system has a core and a set of selectable features. Examples of features include pulling events, dispatching strategies, event payload types, event correlation and filtering, and event profile.

Alice (Eichberg and Mezini, 2005) is a middleware that proposes the combination of aspects and annotations to implement extra-functional services, such as authentication and sessions. AspectJ2EE (Cohen and Gil, 2004) proposes the use of AOP to implement open, flexible and extensible container middleware. Soares et al. (Soares et al., 2002) 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.

7. Conclusions

In this paper, we have described an implementation of collocation optimization for an aspect-oriented communication middleware system. In the proposed solution, the use of aspects was vital to achieve an implementation of collocation with the following characteristics:

- *Modularization*: As usual in AspectJRMI, the code that supports collocation is completely encapsulated in aspects. In other words, the proposed implementation does not suffer from high degrees of spreading and tangling, as is the case of object-oriented implementation of collocation in most middleware systems.
- *Reconfigurability and customizability*: Accordingly with horizontal decomposition principles, AspectJRMI users can decide if they want or not to incorporate collocation

tests to their systems. If they decide to remove collocation verification from the middleware code – for example, to reduce memory footprint requirements – they just need to remove the collocation aspects from the weaving process.

- *Performance*: As described in Section 5, the performance gains generated by collocation are significant – in many situations collocated calls are almost 100% faster than equivalent calls dispatched through the middleware infrastructure. Such gains happen mainly when the middleware cost is significant when compared with the executing time of the invoked method.

As future work, we intend to investigate the use of aspects to support other features in AspectJRMI, such as security, load balancing, fault tolerance and persistence.

Acknowledgement

This research was supported by a grant from The State of Minas Gerais Research Support Foundation (FAPEMIG).

References

Box, Don, 1997. Essential COM. Addison Wesley.

Cardelli, Luca, Davies, Rowan, 1999. Service combinators for web computing. IEEE Transactions on Software Engineering 25 (3), 309–316.

Cohen, Tal, Gil, Joseph, 2004. AspectJ2EE = AOP + J2EE. 18th European Conference on Object-Oriented Programming. In: LNCS, vol. 3086. Springer-Verlag, pp. 219–243.

Eichberg, Michael, Mezini, Mira, 2005. Alice: Modularization of middleware using aspect-oriented programming. 4th International Workshop on Software Engineering and Middleware. In: LNCS, vol. 3437. Springer-Verlag, pp. 47–63.

Henning, Michi, 2004. A new approach to object-oriented middleware. IEEE Internet Computing 8 (1), 66–75.

Object Management Group. The common object request broker: Architecture and specification revision 3.0.2, December 2002.

Orbacus. Available from: <http://www.orbacus.com>.

ORBit2. Available from: <http://orbit-resource.sourceforge.net>.

Pereira, Fernando Magno, Valente, Marco Tulio, Bigonha, Roberto, Bigonha, Mariza, 2006. Arcademis: A framework for object oriented communication middleware development. Software Practice and Experience 36 (5), 495–512.

Pilhofer, Frank, 1999. Design and implementation of the portable object adapter. Diploma Thesis. Johann Wolfgang Goethe-Universitat.

Pratap, Ravi, 2003. Efficient customizable middleware. Master's thesis, Department of Computer Science and Engineering, Washington University.

Puder, Arno, Romer, Kay, 2000. MICO An Open Source CORBA Implementation. Morgan Kaufmann.

Putrycz, Erik, Bernard, Guy, 2002. Using aspect oriented programming to build a portable load balancing service. 22nd International Conference on Distributed Computing Systems (Workshops). IEEE Computer Society, pp. 473–480.

Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, Aniruddha S. Gokhale. Applying optimization principle patterns to design real-time ORBs. In: 5th USENIX Conference on Object-Oriented Technologies, 1999. pp. 145–160.

Sanjay R. Radia, Graham Hamilton, Peter B. Kessler, and Michael L. Powell. The spring object model. In USENIX Conference on Object-Oriented Technologies, 1995.

Schmidt, Douglas C., Cleeland, Chris, 1999. Applying patterns to develop extensible and maintainable ORB middleware. IEEE Communications 37 (4), 54–63.

Schmidt, Douglas C., Wang, N., Vinoski, S., 1999. Object interconnections collocation optimizations for CORBA. SIGS C++ Report 10 (9).

Siegel, Jon, 2000. CORBA 3 Fundamentals and Programming, 2nd ed. John Wiley and Sons.

Soares, Sergio, Laureano, Eduardo, Borba, Paulo, 2002. Implementing distribution and persistence aspects with AspectJ. 17th ACM Conference on Object-Oriented programming systems, languages, and applications. ACM Press, pp. 174–190.

Valente, Marco Tulio, Tirelo, Fabio, Leao, Diana Campos, Palhares, Rodrigo, 2005. An aspect-oriented communication middleware system. International Symposium on Distributed Objects and Applications. In: LNCS, vol. 3761. Springer-Verlag, pp. 1115–1132, October 2005.

Volter, Markus, Kircher, Michael, Zdun, Uwe, 2005. Remoting patterns foundations of enterprise, internet and realtime distributed object middleware. John Wiley and Sons.

Wollrath, A., Riggs, R., Waldo, J., 1996. A distributed object model for the Java system. In: 2nd Conference on Object-Oriented Technologies and Systems, pp. 219–232.

Zhang, Charles, Jacobsen, Hans-Arno, 2003. Quantifying aspects in middleware platforms. 2nd International Conference on Aspect-Oriented Software Development. ACM Press, pp. 130–139.

Zhang, Charles, Jacobsen, Hans-Arno, 2003. Refactoring middleware with aspects. IEEE Transactions Parallel and Distributed Systems 14 (11), 1058–1073.

Zhang, Charles, Jacobsen, Hans-Arno, 2004. Resolving feature convolution in middleware systems. 19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. ACM Press, pp. 188–205.