Journal of Universal Computer Science, vol. 10, no. 7 (2004), 824-842
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04

Tactics for Remote Method Invocation

Fernando Magno Quintão Pereira
(Computer Science Department, Federal University of Minas Gerais, Brazil
fernandm@dcc.ufmg.br)

Marco Tulio de Oliveira Valente
(Computer Science Department, Catholic University of Minas Gerais, Brazil
mtov@pucminas.br)

Wagner Salazar Pires
(Computer Science Department, Catholic University of Minas Gerais, Brazil
wagner@pucmg.br)

Roberto da Silva Bigonha
(Computer Science Department, Federal University of Minas Gerais, Brazil
bigonha@dcc.ufmg.br)

Mariza Andrade da Silva Bigonha
(Computer Science Department, Federal University of Minas Gerais, Brazil
mariza@dcc.ufmg.br)

Abstract: Conventional object oriented middleware platforms rely on the notion of remote interfaces to describe distributed services. This notation is very similar to the one currently used in centralized systems, which increases the productivity of programming. This paper is founded in the observation that remote interfaces foster a programming model that ignores the differences between local and remote interactions. This can result in distributed applications with poor performance, that are not robust to failures, and that can not scale beyond local networks. Therefore, we propose that remote interfaces should be accompanied by the specification of tactics that deal with typical events in distributed computing, such as concurrency, partial failures and high latencies. The paper proposes a tactics definition language and describes the implementation of a middleware system that supports this language.
Key Words: Middleware, tactics definition language, distributed programming
Category: D.1.5, D.3.3, C.2.4

1 Introduction

In the last decade, distributed systems engineers have often relied on middleware platforms to increase their productivity. Residing between the operating system and distributed applications, middleware systems provide abstractions that hide from application developers several details inherent to distributed programming, such as network communication primitives, data marshalling and unmarshalling, failure handling, heterogeneity, service lookup and synchronization. At the present time, object-oriented systems – like CORBA [Object Management Group, 2000], and Java RMI [Wollrath et al., 1996] – are the most common middleware platforms. In such systems, developers invoke methods on remote objects using the same syntax of local invocations; therefore, code to handle distributed communication looks similar to code that handles communication in centralized systems.

In commercial middleware systems, remote services are specified using the concept of interfaces. CORBA defines a specific language, called IDL (*Interface Definition Language*), to describe the interfaces of remote objects. CORBA also defines bindings between IDL and general purpose programming languages, such as C, C++ and Java. For this reason, CORBA is often classified as a language neutral middleware. Similarly, remote services in Java RMI are specified using the standard concept of interfaces of Java. Remote interfaces in this system must extend the `java.rmi.Remote` interface and their methods must throws a `java.rmi.RemoteException`. Thus, systems like Java RMI and CORBA strive to unify the specification of local and remote objects. The idea is to provide high level abstractions – like remote method invocations – that make distributed programming as simple as conventional programming.

However, as pointed in [Waldo et al., 1997], there are fundamental differences between interactions of distributed and non-distributed objects with respect to latency, concurrency, partial failure and the model of memory access. Regarding latency, the difference between local and remote invocations is around four and five orders of magnitude, and, if taken into consideration the Internet or wireless networks, this gap is even greater. Also, the occurrences of failures is far more common in distributed systems than in centralized ones. Moreover, in distributed applications failures are worse because there is not a global state that can be queried in order to discover the type and the source of errors. Finally, distributed objects are intrinsically concurrent, frequently having to handle simultaneous calls.

This paper is founded in two observations about the use of remote interfaces in conventional middleware platforms:

- Remote interfaces provide a high level notation to describe distributed services. This notation is very similar to the one currently used in centralized systems, which increases the productivity of programming.

- Remote interfaces foster a programming model that ignores the differences between local and remote interactions. This can result in distributed applications with poor performance, non tolerant to failures, and that can not scale beyond local and small networks.

In order to clarify the second observation we can rely on the use of remote interfaces in Java RMI. Interfaces do not allow the application developer to specify the reliability level of methods, their priority, or what kinds of enhancements that can be used by the middleware in order to improve invocations' performance and fault tolerance. For example, methods that neither cause side effects nor throw exceptions could take benefit from a cache in order to avoid unnecessary accesses to the network. As another example, the declaration of remote methods in Java RMI only specifies that communication failures should raise an exception. It is not possible, for example, to specify that a secondary server should be contacted, if the primary service provider is not accessible. Also, it is not possible to define that calls to a service should be dispatched to more than one server in order to provide load balancing or to increase performance. Moreover, in Java RMI programmers can not decorate remote invocations with extra processing, such as logs, buffers and timeouts.

In this paper, we argue that *remote interfaces should be preserved as the basic notation for the specification of remote services* in object oriented middleware systems, since they provide a high level of abstraction to programmers not familiar with the details of distributed computing. However, we also argue that *remote interfaces should be accompanied by a specification of the tactics used to deal with phenomena typical of distributed settings*, such as concurrency, partial failures and high latency. A set of tactics specifications associated to a remote interface can be used to define aspects such as the following:

- the semantics used to dispatch remote invocations (e.g. best-effort, at-least-once, at-most-once, etc). The invocation semantics determines the level of reliability the underlying middleware system provides to application developers regarding the execution of remote methods;

- extra capabilities that can be added to remote methods in order to enhance their non-functional aspects, such as fault tolerance and performance. Examples of enhancements that may be added to remote operations include caches, buffers and log generators, among others;

- the distribution of priorities among different methods. Generally remote objects process calls in a first in, first out fashion; however, there are situations when it is desirable to reduce the waiting time of critical operations giving them higher priorities;

- the existence of more than one remote object providing the same service. For example, in an over-provisioned environment, programs should be able to specify that servers be contacted concurrently (to increase performance), sequentially (to increase fault tolerance) or non-deterministically (to provide load balancing).

The remaining of this paper is organized as follows: Section 2 defines and explains a tactics definition language for object oriented middleware platforms.

Tactics definition

$$T \rightarrow Service \mid Method \mid Priority \mid T_1\ T_2$$

Service

$$Service \rightarrow Id = Id_1 \mid Id = Host/Id_1$$

tactics

$$\begin{aligned} Method &\rightarrow Id = S.D.I \\ S &\rightarrow S_1 \text{ `|' } S_2 \mid S_1\ ?\ S_2 \mid S_1 > S_2 \mid (S_1) \mid Id \\ D &\rightarrow \lambda \mid D_1 + D_2 \mid \texttt{Asynch}(time) \mid \texttt{Cache}(Number) \mid \texttt{Log}(String) \mid \texttt{Timer}(Number) \\ I &\rightarrow \mathrm{OneWay}() \mid \mathrm{TwoWay}() \mid \mathrm{AtLeastOnce}(Number, Number) \\ &\quad\ \mathrm{AtMostOnce}(Number, Number) \end{aligned}$$

Distribution of priorities

$$Priority \rightarrow .Number\ @\ Id$$

Regular expressions

$$\begin{aligned} Id &\rightarrow letter(letter|digit)* \\ Number &\rightarrow digit+ \\ Host &\rightarrow String \text{ `:' } Number \mid String \\ letter &\rightarrow [\mathrm{A–Z}]|[\mathrm{a–z}] \\ digit &\rightarrow [0\text{–}9] \\ String &\rightarrow any\ sequence\ of\ characters \end{aligned}$$

Figure 1: Tactics Definition Language

This section also shows an example of tactics usage. Section 3 describes a middleware system that implements the proposed tactics system. Section 4 compares the introduced approach with similar works. Finally, Section 5 concludes the paper.

2 A Tactics Definition Language

Tactics describe procedures to customize and adapt applications to events typical of distributed computing, such as partial failures, higher latencies, synchronization and concurrency. In this section we propose a tactics definition language for object oriented middleware systems. The general structure of this language is described in Figure 1 by means of a BNF-like notation.

The remainder of this section describes the semantics and gives a rationale for the proposed tactics language.

2.1 Services

In order to define a set of tactics, programmers should declare the service providers that can be contacted to process remote invocations. The declaration of a service provider aims at informing its location, which is given by a host name (or IP address) and by a name that uniquely identifies the service in the host. If the host name is not given, it is assumed that the service is located in the local host. The following example declares a service located in the host `turmalina.dcc.ufmg.br` and named `foo`. In the remaining of the tactics specification file where this declaration appears, whenever necessary to refer to this service, the name `srv1` should be used instead.

```
srv1 = turmalina.dcc.ufmg.br/foo
```

2.2 Tactics

Tactics are specifications of procedures to be adopted by the middleware platform in order to deal with particularities of the distributed environment, such as high latency and communication crashes. The methods declared in a remote interface can be associated to different tactics specifications.

The declaration of the set of tactics bound to a remote method may be divided into three parts: the declaration of service combinators, the specification of invocation decorators and the definition of a reliability level. The first part concerns the choice of the service provider that will process the remote call, and the second part defines the chain of enhancements that will be added to the remote invocation. Predefined enhancements include the use of caches, buffers, logs, timeouts and support for asynchronous calls. Finally, the last part defines the remote call semantics, for example: best-effort, at-most-once and at-least-once.

The parts that constitute a tactics specification can be regarded as orthogonal sets. Therefore, any service combinator may be combined with any reliability level. Furthermore, invocation decorators may be aggregated to remote methods, independently of the other types of tactics bound to them. The only restriction concerns the reliability level known as *one-way*, as discussed in Section 2.3.

2.2.1 Service Combinators

Service combinators rely on the existence of more than one remote object providing the same services in order to give remote operations support to load balancing, fault tolerance, and improvements in performance. Service combinators are the first kind of tactics specification to be handled. For example, if alternative execution is combined with the at-most-once reliability level, described in Section 2.3, the same server may be contacted several times before the next available remote object be activated. The available service combinators are:

S_1 **?** S_2 **Non-deterministic choice** Exactly one of the service providers, S_1 or S_2 is non-deterministically chosen to perform the remote invocation. Because invocations are equally divided among the available service providers, this combinator provides support to load balancing.

$S_1 \mid S_2$ **Concurrent execution** Both service providers, S_1 and S_2, are concurrently activated to perform the remote invocation. The first answer that arrives is returned as the result of the call and the other is discarded. Thus, this combinator is used to optimize the response time of remote invocations.

$S_1 > S_2$ **Alternative execution** First, the service provider S_1 is invoked. If for some reason a result is not obtained from this service, the invocation is dispatched to S_2. Hence, this combinator provides support to fault tolerance.

2.3 Request Reliability

Since networks are subjected to various kinds of failures, different levels of reliability can be provided by the protocols used to transmit remote messages in object oriented middleware systems. Unfortunately, there is a trade-off between the reliability and the performance of such protocols. Therefore, a remote invocation that does not require high degrees of reliability can take benefit of a simpler delivery protocol.

In the proposed tactics definition language, it is possible to define the level of reliability required in the invocation of each method of a remote service. The available reliability levels are the following:

One-Way This level does not guarantee the execution of remote invocations and clients are not notified when the invocation fails. After a one-way remote invocation is transmitted to the underlying middleware system, the calling thread continues its execution. Thus, one-way invocations should not have return values nor raise exceptions. This is the lowest level of reliability provided, and the one with the lowest implementation overhead. Because one-way calls do not generate any response from service providers, there is no point in using this strategy with tactics such as caches, timers or asynchronous calls.

Two-Way This level does not guarantee the execution of remote invocations but an exception is raised when the invocation fails. The calling thread remains blocked waiting the result of the call or a timeout. In case of timeout, an exception is thrown, and no further processing is performed in order to check if the remote method was executed or not. This level should be used in environments with low error rates, like local networks with stable servers.

At-Most-Once This level does not guarantee the execution of remote invocations. However, in case of failure, the invocation is automatically retransmitted a certain number of times. If all retransmissions fail, an exception is raised. It is also assured that an invocation will not be processed two or more times. This level should be used in environments where failures are common, such as the Internet and wireless networks.

At-Least-Once This level guarantees the execution of remote invocations, possibly more than once. Multiple executions happen when the results of remote invocations are successively lost. In this case, the middleware retransmits the invocation, which might result in extra processing of the remote method. This is the highest level of reliability, although it is not recommended when remote methods have side-effects. Moreover, the calling thread will remain blocked in case of continuous unavailability of the remote service.

2.4 Invocation Decorators

Invocation decorators define extra behaviors that are transparently inserted into the dispatching flow of remote operations. Invocation decorators can also be combined in order to create chains of responsibilities that are attached to remote methods.

The proposed tactics definition language supports the following predefined invocation decorators:

- `Cache`(*size*): caches may be used to store the results of remote invocations in an attempt to reuse them later when the same invocation is triggered again. A cache is particularly useful when the associated method does not generate side-effects. In this case, it trades space for response time. The parameter of this kind of decorator represents the size of the cache, i.e., the maximum number of bytes that can be stored on it.
- `Timer`(*time*): this decorator allows to define time limits for the execution of remote calls. Despite its reliability level, if a remote call is not performed in the specified *time* parameter, its execution is aborted and an exception is raised. Such decorator is useful in applications that can not tolerate unpredictable delays, such as the real-time systems.
- `Log`(*file*): creates a log containing informations about remote invocations. The *file* parameter indicates the name of the log file.
- `Asynch`(*time*): this decorator is used to support asynchronous remote invocations. When an asynchronous invocation is requested, an object of the type `Future` is created and returned to the client thread (that continues its own processing). A separate thread is created to deal with the remote invocation. Later, when the result of the invocation becomes available, it is inserted into the future object. The client thread must poll this object to check for the result. The decorator parameter specifies the maximum amount of time the client can wait before the result is available. Finished that time, if the call has not being processed, an exception object is inserted into the future.

2.5 Priorities

In conventional middleware systems, all remote invocations are given the same priority. However, in order to increase the overall performance of the system, it may be useful to change the priority of particular invocations. For example, if a method takes a considerable time to finish and the calling thread supports delays on that operation, it is recommended to assign a small priority to it. As a consequence, the server object will postpone the execution of this operation, while quickly processing other simpler calls.

The proposed tactics definition language supports the assignment of priorities with the following syntax: $n@Id$, where n is a number between 0 and 1 and Id is a method name. This command reduces the priority of the specified method to the value $n \times d$, where d is the default priority. The lowest is the value $n \times d$, the lowest is the invocation priority of the associated method. Therefore, in the proposed tactics language, developers can not increase the priority of a remote operation; only reduce it.

2.6 An Example of Tactics

In order to better illustrate how tactics may be used to determine some of the non-functional aspects of remote methods, this section presents a translation service whose operations have been associated to different tactics. This example is based on the tactics-based middleware platform described in Section 3.

The translation service provides three remote operations: `ip_word` (translates single words), `ip_paragraph` (translates sentences containing up to four hundred characters) and `ip_text` (translates text files). The remote interface of these operations is the following:

```
interface IP_Translator extends Remote {
  public String ip_word(String w)
    throws ArcademisException, NotWordException;
  public String ip_paragraph(String p)
    throws ArcademisException, TextTooBigException;
  public MarshalableFile ip_text(MarshalableFile f)
    throws ArcademisException;
}
```

This interface uses a Java syntax, and classes like `ArcademisException` and `Remote` that are provided by the middleware introduced in Section 3. We consider that a client application can take benefit from the following tactics set when invoking the translation methods:

```
turmalina = turmalina.dcc.ufmg.br/glossary;
diamante = diamante.dcc.ufmg.br/glossary;
sirius = sirius.inf.pucminas.br/ip_translator;
```

```
ip_word = (turmalina | diamante).Cache(2048)+Timer(1000).
          AtLeasOnce(8,100);
ip_paragraph=((turmalina?diamante)>sirius).AtMostOnce(12,100);
ip_text = (turmalina > diamante > sirius).Asynch(0).TwoWay();

.8@ip_paragraph
.6@ip_text
```

This tactics specification assumes that the translation service is available on three different hosts: `turmalina`, `diamante` and `sirius`. The following tactics are associated to invocations of remote methods:

- `ip_word`: in order to reduce the response time, invocations of this method are concurrently transmitted to the services available on `turmalina` and `diamante`. A cache is associated to `ip_word` invocations, since the client application should request later the translation of the same word. The cache is augmented with a timer, so that if the method is not executed within one second (1000 milliseconds), the operation will be aborted. The reliability level is at-least-once: the parameters tell the middleware to perform up to eight attempts of contacting the service provider, being 100 the time interval, in milliseconds, between successive calls.

- `ip_paragraph`: on a first trial, invocations of this operation are non-deterministically distributed among services on `turmalina` and `diamante`. If both services fail to provide an answer, the remote service on `sirius` is contacted. Neither a cache, nor any other type of decorator is used in order to aggregate extra capabilities to this method. The reliability level defined for this type of invocation is the at-most-once semantics. The parameters determine that up to 12 attempts of contacting the service provider will be made in time intervals of 100 milliseconds until an answer is available. In addition, the server will not process repeated calls.

- `ip_text`: a translation of a file can demand a reasonable computation effort. Thus, this operation is asynchronous in order to release the calling thread while the result of the translation does not arrive. The operation is firstly dispatched to the service known as `turmalina`. If this invocation fails, a new attempt is made on `diamante`, and, if this object is also not available, the method is finally invoked on `sirius`. The zero parameter in the `Asynch` decorator specifies that the thread in charge of the call can only be aborted by the underlying middleware (and not by a timeout). The `MarshableFile` must implement the `Future` interface in order to allow clients to check for the result of the call. Because three different service providers can be successively contacted in order to process this method, it is possible to define its reliability level as two-way.

Invocations of `ip_paragraph` and `ip_text` have their priority reduced respectively to 80% and 60% of the default priority. Since the priority of `ip_word` is not declared, this method is assigned the highest priority value.

3 Implementation

This section introduces Aries[1], a middleware platform that implements the tactics system proposed in this paper. This platform has been implemented as an instance of Arcademis [Pereira, 2004], a framework for middleware development.

3.1 Implementation of the Invocation Policy

In distributed systems, client and server objects may be located in different address spaces; so, it is necessary to provide client applications with local representatives of the remote objects. These local representatives are called *stubs*. The stub acts as a proxy, having the same interface as the remote object it represents. Therefore, whenever the client invokes a remote operation, it is actually invoking one of the methods of the stub. The stub is responsible for marshaling the parameters of remote calls and unmarshalling their return values, if they exist; however, in Arcademis, the component that actually sends the invocation request across the network is called *invoker*. Invocation decorators and reliability levels are implemented by invokers; hence, in order to allow different methods to use different tactics, the stub must have access to a collection of such components. Arcademis makes such collection available by means of the *invoker factory*.

The partial description of a remote method invocation, as it is performed in Arcademis/Aries, is given by the collaboration diagram in Figure 2. According to that scheme, `ip_word` is a remote method that the client application is invoking on `stb`, a stub of the `Translator_Stub` type. Stubs, in Aries, are subclasses of `SuperStub`. `Stb` marshals the call parameters and passes them to its superclass, but, before doing this, it determines the type of components that will be produced by the invoker factory. Upon receiving a request, the `SuperStub` implementation gets from the invoker factory the component that will perform the invocation according to the chosen strategy. The invoker sets up a connection with the server and sends to it an object of the `Message` type, that holds the call descriptor. Details of these final procedures are not exhibited in Figure 2.

3.2 Implementation of the Reliability Level and Scheduling Policy

The implementation of the `Invoker` component used to carry on a remote method invocation determines the call's reliability level, as discussed in Section 2.3. However, for the purpose of ensuring a given invocation semantics,

[1] Aries is an acronym for *Another Remote Invocation System.*

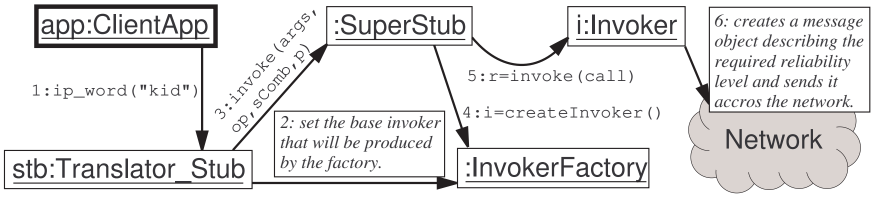


Figure 2: Partial view of the invocation path in the client side.

some processing is also necessary in the server side of the middleware platform. For example, in order to implement the at-most-once level of guarantee, the server has to keep a list of identifiers of already processed calls. In Aries, the entity responsible for receiving invocation requests from the network layer is the `RequestReceiver` component. Due to Aries' design, it is not necessary to provide the `RequestReceiver` with a separate clause for handling every kind of reliability level. Arcademis defines the middleware communication protocol by a set of *messages*: objects that implement the `Message` interface. Messages are handled by means of the *Command* design pattern [Gamma et al., 1994]. The `Message` interface defines an `execute()` operation, whose implementation determines all the necessary processing to assure the reliability level promised by the invoker implementation.

The collaboration diagram presented in Figure 3 depicts the path of a remote method invocation after it is received at the server side. The invoker determines different reliability levels producing different types of messages to carry remote calls. Once one of these messages is received, the `RequestReceiver` invokes the `execute` operation on that object. The implementation of this method must ensure that the invocation parameters will reach the remote object that is providing the service requested by the call, and that the correct reliability level will be guaranteed.

At the server side, remote invocations are ordered by the `Scheduler` component. The `Scheduler` and the `RequestReceiver` are active objects, that is, they execute on their own control threads. Before issuing a remote call, the stub implementation assigns to its descriptor a default priority value, or a specific one, according to what is determined in the tactics specification file. Each message the `Scheduler` receives is inserted into a priority queue. The scheduling thread continuously removes invocation requests from the queue, and delivers them to the remote object that is responsible for their execution.

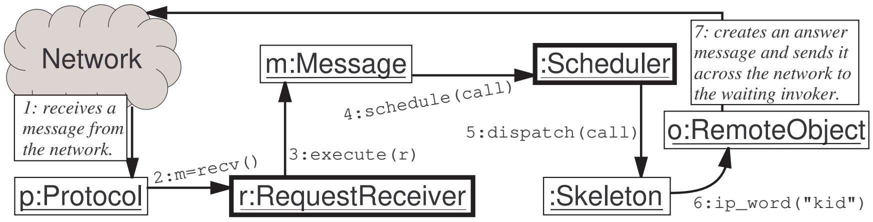


Figure 3: Partial view of the invocation path in the server side.

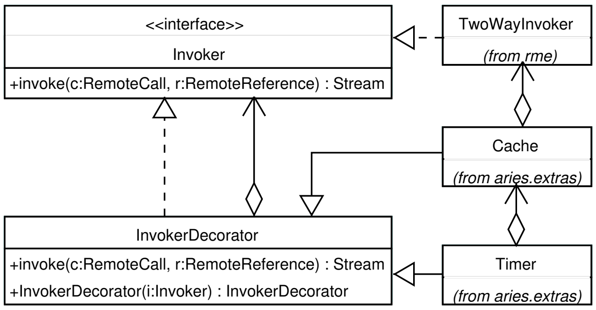
3.3 Implementation of Invocation Decorators

The extra responsibilities described in Section 2.3 are inserted into the invocation path of remote methods by means of *invoker decorators*. Decorator is a design pattern [Gamma et al., 1994] which characterizes enclosing objects that modify the behavior of other objects while avoiding the generation of complex chains of inheritance. A invoker decorator is a subclass of `Invoker`, and, in addition, has an attribute of the `Invoker` type, which is the enclosed object. The decorator may overwrite the `invoke` method in order to intercept its parameters, and perform some processing on them, before passing these arguments to the intercepted method. Because the invoker decorator has also the `Invoker` type, subclasses of it can be assembled together in order to compose chains of extra functions that may be added to the same invoker. The class diagram shown in Figure 4 (a) depicts an example of a chain of decorators aggregated to a base invoker. These decorators implements the cache and timer tactics discussed in Section 2.4.

Figure 4 (b) presents the `invoke` method of a decorator that adds a cache to the invoker. When this decorator receives an invocation request, it firstly verifies if that call has already been processed. If so, it returns to the caller the operation's return value saved in the cache during the first call; otherwise the decorator sends the request to the enclosed invoker and stores its result when it is available. In the figure, class `OrbAccessor` represents a façade for creating objects, and `Stream` encapsulates serialized data, such as the arguments of remote calls.

3.4 Automatic Generation of Stubs

In Aries, the tactics used to perform a remote method invocation are defined by the stub implementation; therefore, for each of the stub's methods, customized code must be produced. Stubs in Aries are automatically generated by a tool named `AriesC`, which produces Java source code from a remote class and a tactics specification file, as depicted in Figure 5 (a). The remainder of this section



(a)

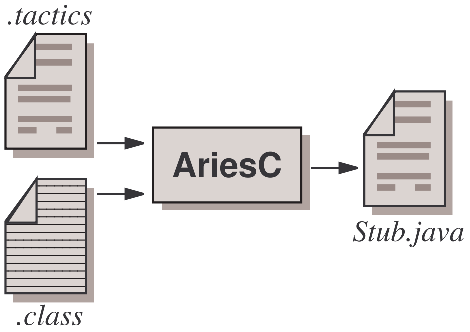
```
// method from CacheDecorator.java
public Stream invoke(RemoteCall c)
throws NetworkException {
  Stream args= c.getArguments();
  if(this.table.containsKey(args)) {
    Stream r= OrbAccessor.getStream();
    r.fill((Stream)this.table.get(args));
    return r;
  } else {
    Stream r= super.invoker.invoke(c);
    Stream bck= OrbAccessor.getStream();
    bck.fill(r);
    table.put(args, bck);
    return r;
  }
}
```

(b)

Figure 4: (a): Invoker decorators. (b): Decorator that adds a cache do the `Invoker`.

presents and discusses some examples of automatically generated code obtained from the tactics specification file exposed in Section 2.6.

`AriesC` produces for each stub an initialization routine, which is called `findServiceProviders`. Such a method allows to bound the stub to the service providers it has to represent in the client address space. The service providers are declared in the tactics specification file, and, for each of them, a remote reference must be obtained from the discovery agency and attached to the stub. Aries also allows dynamic binding, that is, it is possible to assign a remote object to a initialized stub; however, such object will be used only if no other service provider has been specified by tactics. An instance of the initialization method, produced from the example of Section 2.6, is presented in Figure 5 (b). `AriesNaming` is the interface for the discovery agency provided by the Aries system, and code for error recovery is not shown, due to space restrictions.



(a)

```
  private boolean ready = false;
  public findServiceProviders() {
    try {
      RemoteReference r1 = AriesNaming.
      lookup("turmalina.dcc.ufmg.br/glossary");
      super.attach(r1);
      RemoteReference r2 = AriesNaming.
      lookup("diamante.dcc.ufmg.br/glossary");
      super.attach(r2);
      RemoteReference r3 = AriesNaming.
      lookup("sirius.inf.pucminas.br/ip_translator");
      super.attach(r3);
      ready = true;
    } catch (NotBoundException nbe) {
    } catch (MalformedURLException me) {}
}
```

(b)

Figure 5: (a): Generation of Stubs in Aries. (b): Example of discovery routine.

Figure 6 and 7 exhibit code automatically generated for two of the methods declared in the interface of the translation system discussed in Section 2.6. Parts of these methods have been produced from the tactics specification file that accompanied that interface. Before effectively issuing a call, by means of the `invoke` method, the stub has to define the implementation of the invoker that will be in charge of performing the operation[2]. In Figure 6, this is done by the commands from line 8 to 12. Firstly, the stub obtains a reference to the invoker factory; then it determines a list of decorators that must be aggregated to the

[2] It is possible to improve performance if the invokers are created during the stub initialization, and assigned to the factory before method invocations. The approach presented in Figures 6 and 7, in which new invokers are instantiated before each call, has been chosen because it makes clearer how the stub implements the tactics described by the application developer.

```
1: public String ip_word(String param0)
throws NotWordException, ArcademisException {
2:   if(!ready)
3:     findRemoteReferences();
4:   Stream args = (Stream)OrbAccessor.getStream();
5:   args.write(param0);
6:   int op = 1;
7:   String servers = "(turmalina|diamante)";
8:   InvokerFactory fc = ORB.getInvokerFactory();
9:   fc.removeDecorators();
10:  fc.insertDecorator(new Cache(2048));
11:  fc.insertDecorator(new Timer(1000));
12:  fc.setComponent(new AtLeastOnce(8,100));
13:  this.setInvokerFactory(fc);
14:  int p = 1000;  // the operation priority.
15:  Stream future = invoke(args, op, servers, p);
16:  if(future.isException()) {
17:    // handle remotely rised exceptions here
18:  }
19:  return future.readString();
20:}
```

Figure 6: Example of automatically generated stub methods.

next components the factory will produce. Finally the stub defines the base implementation of the invoker to be generated. During code generation, `AriesC` considers invocation decorators and invoker definitions present in the tactics specification file as Java constructor methods.

The priority of invocations and the pattern for choosing service providers are also defined by the stub. The priority is defined in the lines 14 and 12 of Figures 6 and 7 respectively. In the Aries system, priority values range from 0, the lowest value, to 1000. Service combinators are described by means of strings, which are parsed in the `invoke` method (Figure 6 – line 15) of the `aries.SuperStub` class. This method receives four parameters: the serialized arguments of the invocation, the operation identifier, the descriptor of service combinators (Section 2.1) and the invocation priority (Section 2.5). The code presented in Figures 6 and 7 has been simplified due to space constraints: routines for handling exceptions thrown in the server side have been omitted.

4 Related Work

The following systems or research projects are related to the tactics system proposed in this paper.

CORBA CORBA [Object Management Group, 2000] supports some of the tactics proposed in this paper. For example, methods can be qualified as

```
public String ip_paragraph(String param0)
throws TextTooBigException, ArcademisException {
  if(!ready)
    findRemoteReferences();
  Stream args = (Stream)OrbAccessor.getStream();
  args.write(param0);
  int op = 2;
  String servers = "(turmalina?diamante)>sirius";
  InvokerFactory fc = ORB.getInvokerFactory();
  fc.removeDecorators();
  fc.setComponent(new AtMostOnce(12,100));
  this.setInvokerFactory(fc);
  int p = 800;  // the operation priority.
  Stream future = invoke(args, op, servers, p);
  if(future.isException()) {
    // handle remotely rised exceptions here
  }
  return future.readString();
}
```

Figure 7: : Example of automatically generated stub methods.

one-way, which means they are called using the one-way semantics described in Section 2.3. Also, meta-programming mechanisms, such as interceptors and smart proxies, can be used to implement tactics such as invocation decorators and service combinators. However, these mechanisms work by changing the default behavior of the middleware internals; thus, they provide a very low level of abstraction. On the other hand, we argue that our tactics definition language allows easy expression of common strategies for handling events typical of distributed systems. Moreover, tactics do not have a meta-programming or reflective semantics; hence, tactics specification do not involve reification. In our language, tactics are expressed at the same level of abstraction, for example, than remote interfaces in IDL.

TAO TAO [Schmidt and Cleeland, 1999] is an extensible and maintainable middleware based on CORBA. TAO uses the service configurator design pattern to support configuration of several aspects of the middleware platform. A configuration file defines internal strategies of the middleware like thread policies, request demultiplexing, scheduling and connection management. At startup time, the configuration file is loaded and the selected strategies are applied. Thus, TAO supports the configuration of global aspects of the middleware engine. On the other hand, tactics provide support for fine-grained customizations that are related to particular remote services. Moreover, configuration in TAO affects mainly components placed on the server side of distributed applications. On the other hand, tactics mostly support the customization of client side aspects of remote method invocation.

Service Combinators Service combinators were first proposed in [Cardelli and Davies, 1999] to define strategies for handling failures and slow communications links when retrieving content on the Web. They were later incorporated in the WebL language [Kistler and Marais, 1998].

Aspect Oriented Programming Aspect-oriented programming (AOP) [Kiczales et al., 1997] is an alternative technology for separation of concerns in software development. AOP languages, such as AspectJ [Kiczales et al., 2001], provide abstractions to modularize crosscutting concerns of a system, and permit to *weave* aspects with conventional code. AspectJ provides pattern mechanisms (pointcuts) to capture events (joinpoints) in the execution flow of conventional programs. Extra code, called advices, can be defined to run when pointcuts are reached. Similar to aspect languages, tactics define strategies that crosscuts the traditional vertical decomposition structure of distributed systems. However, the language proposed in this paper does not present a set of operators and constructions as complete as, for example, AspectJ does. This is explained by the fact that AspectJ is a general-purpose aspect language while our tactics language can be considered a domain-specific aspect language that targets the customization of remote invocations in object oriented middleware.

Chroma A previous tactics based remote execution system, called Chroma, is discussed in [Balan et al., 2003]. The main purpose of tactics in Chroma is to describe how applications can be dynamically partitioned for execution in mobile computing environments. Chroma uses three techniques in order to select tactics: resource prediction, resource monitoring and user guidance. For example, suppose a Chroma based language translator system for mobile computing. If the available bandwidth is sufficiently large, the system would automatically adopt a more resource demanding service in order to perform translations; a simpler service would be used otherwise. Since tactics in Chroma are selected dynamically, they are more flexible than the tactics system presented in this paper. On the other hand, our tactics definition language supports several abstractions that are not available in Chroma, like reliability levels, invocation decorators and priorities.

DADO The DADO (Distributed Adaplets for Distributed Objects) [Wohlstadter et al., 2003] system targets the implementation of non-functional requirements in distributed and heterogeneous systems. DADO services are expressed by means of adaplets that: (i) are modeled in an extension of IDL (called DADO IDL); (ii) may be implemented in any language and (iii) are attached to stubs and skeletons. Adaplets support the definition of methods that are bound to application objects using a pointcut language similar to AspectJ. The system also provides abstractions to coordinate computations that encompass both client and server adaptlets. Similar to tactics, adaplets do not involve reification of middleware state (as happen with interceptors in CORBA). On the other hand, DADO is a general purpose programming system while tactics were designed in order to make remote invocations more robust, efficient and reliable.

QuO Quality Objects (QuO) [Zinky et al., 1997] is a framework that adds quality of service support to CORBA and Java RMI. Similarly to Chroma and Aries, QuO extends a standard interface definition language with a meta-language that supports the specification of a contract between clients and servers concerning quality of service requirements. QuO permits the definition of a set of regions that are characterized by different QoS requirements. The middleware will react when the QoS status moves between such regions.

5 Conclusions

This paper has proposed a domain specific language to define tactics adopted in remote method invocations to handle particular characteristics of distributed environments, such as communication failures and high latencies. The main motivation for the proposed system is the fact that centralized and distributed environments present a number of differences in terms of latency, reliability and concurrency. This makes the representation of remote methods adopted by traditional middleware platforms, such as CORBA and Java RMI, unsatisfactory. Therefore, this paper contributes to increase the flexibility of object-oriented middleware systems in the following ways:

- it has proposed a set of tactics to customize non-functional aspects of remote method invocations;
- it has described the implementation of Aries, a middleware that implements the proposed tactics definition language;

It should be pointed that the tactics system proposed in this paper does not constitute a closed group: other tactics can be incorporated into it, based on the needs of application developers. Finally, the code of Arcademis and Aries can be downloaded from `http://www.dcc.ufmg.br/llp/arcademis`.

Acknowledgments
This paper was developed as part of a research project sponsored by FAPEMIG (process CEX 488/2002).

References

[Balan et al., 2003] Balan, R. K., Satyanarayanan, M., Park, S., and Okoshi, T. (2003). Tactics-based remote execution for mobile computing. In *1st International Conference on Mobile Systems, Applications, and Services*, pages 273–286. USENIX.

[Cardelli and Davies, 1999] Cardelli, L. and Davies, R. (1999). Service combinators for web computing. *Transactions on Software Engineering*, 25(3):309–316.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, volume 2072, pages 327–355. Springer Verlag.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer Verlag.

[Kistler and Marais, 1998] Kistler, T. and Marais, H. (1998). WebL - a programming language for the Web. *Computer Networks and ISDN Systems*, 30:259–270.

[Object Management Group, 2000] Object Management Group (2000). The Common Object Request Broker: Architecture and Specification (version 2.4).

[Pereira, 2004] Pereira, F. M. Q. (2004). Arcademis: Um arcabouço para construção de sistemas de objetos distribuídos em java. Master's thesis, Universidade Federal de Minas Gerais.

[Schmidt and Cleeland, 1999] Schmidt, D. and Cleeland, C. (1999). Applying Patterns to Develop Extensible and Maintainable ORB Middleware. *IEEE Communications Magazine – Special Issue on Design Patterns*, 37(4):54–63.

[Waldo et al., 1997] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). *A Note on Distributed Computing*, pages 49–64. Springer-Verlag.

[Wohlstadter et al., 2003] Wohlstadter, E., Jackson, S., and Devanbu, P. (2003). DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. In *25th International Conference on Software Engineering*, pages 174–186. IEEE Computer Society.

[Wollrath et al., 1996] Wollrath, A., Riggs, R., and Waldo, J. (1996). A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX.

[Zinky et al., 1997] Zinky, J., Bakken, D., and Schantz, R. (1997). Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):1 – 20.