SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **00**:1–18 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

SP&E

Arcademis: A Framework for Object Oriented Communication Middleware Development

F. M. Q. Pereira[1‡], M. T. O. Valente[2,∗§], R. S. Bigonha[1‡] and M. A. S. Bigonha[1‡]

[1] *Department of Computer Science, Federal University of Minas Gerais - UFMG, Belo Horizonte, MG, 31270-010, Brazil*
[2] *Department of Computer Science, Catholic University of Minas Gerais - PUC Minas, Belo Horizonte, MG, 30535-610, Brazil*

SUMMARY

This paper presents Arcademis, a Java-based framework for communication middleware development. Arcademis consists of a set of abstract classes, interfaces and concrete components that define the general architecture of middleware systems. Its main objective is to support the implementation of non-monolithic and easily configurable middleware platforms. Arcademis can be used by middleware developers to deploy systems that meet the requirements of a particular network or technology. Instances of Arcademis can also be customized by distributed systems engineers to meet the requirements of a particular application. For example, new transport protocols, connection management policies, authentication algorithms or invocation semantics can be easily configured in middleware platforms derived from Arcademis. In order to illustrate the use of the framework, the paper describes the RME system, a middleware derived from Arcademis that adds a remote method invocation service to the CLDC configuration of Java 2 Micro Edition (J2ME).

KEY WORDS: Frameworks, communication middleware, distributed systems.

‡E-mail: {fernandm,bigonha,mariza}@dcc.ufmg.br
§E-mail: mtov@pucminas.br
∗Correspondence to: Marco Túlio Valente, Department of Computer Science, PUC Minas, Belo Horizonte, MG, 30535-610, Brazil
Contract/grant sponsor: CNPq, FAPEMIG

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 encapsulate several details inherent to distributed programming, such as network communication primitives, data marshalling and unmarshalling, failure handling, heterogeneity, service lookup and synchronization [6, 22]. At the present time, object-oriented middleware – such as CORBA [16] and Java RMI [26] – are the most common platforms. In such systems, developers invoke methods on remote objects using the same syntax of local invocations; therefore, interactions between local and remote processes seem to coexist in the same address space.

Since the first systems, object oriented middleware platforms have always been designed in order to make object locations transparent to architects of distributed applications, i.e., applications running in standard computers connected by local or enterprise networks. However, in recent years, distributed computing has faced many changes. There are multiple categories of computing devices (sensors, cell phones, PDAs, multicomputers, clusters, grids, etc), multiple network infrastructures (local networks, enterprise networks, Internet, wireless networks, etc), different transport protocols (TCP, UDP, HTTP, etc) and applications with different quality of service requirements (real time systems, multimedia, mobile systems, embedded systems, etc).

Nowadays, it is clear the difficulty of traditional middleware systems in supporting new requirements posed by both end users and middleware developers [5, 8, 12, 23]. Application developers still want to benefit from transparency. However, in some cases, they would like to be able to configure the underlying middleware system in order to optimize an application for a specific environment. For example, they may want to choose a particular transport protocol, a particular authentication algorithm or to assign priorities to remote calls. On the other hand, middleware developers have usually evolved their systems by adding new features. This strategy has resulted in complex and heavyweight platforms. Thus, middleware developers would also benefit from systems that can be configured to fit the particular requirements of an application domain.

This paper presents Arcademis, a Java-based framework that supports the implementation of modular and highly customizable middleware architectures. Arcademis can be used by middleware developers to deploy systems that meet the requirements of a particular network and/or technology. For example, Arcademis has been successfully used to provide a remote method invocation system to J2ME/CLDC [18], the Java technology that targets mobile devices with limited computing resources. Instances of Arcademis can also be adapted by distributed systems developers to meet the requirements of a particular application. For example, new transport protocols, connection management policies, authentication algorithms or invocation semantics can be easily configured in middleware systems derived from Arcademis.

In order to provide configurability and composability, Arcademis makes extensive use of object oriented frameworks and design patterns. A framework is a set of cooperating classes and interfaces that provide a semi-complete application that can be customized by programmers [11]. There are two types of frameworks: black-box and white-box. Frameworks from the first category provide to software engineers a set of concrete components and interfaces that can be composed to implement a specific application. White-box frameworks are constituted by abstract components that the application developer must implement. Arcademis presents characteristics from both categories. Design patterns document recurring solutions to problems in software development [7]. In Arcademis, frameworks and design patterns are applied to promote the implementation of flexible and non-monolithic middleware. As a framework, Arcademis predefines the overall architecture of middleware platforms, so that developers can concentrate on the details of their particular applications. Well-known design patterns, such as singletons, factories, strategies, decorators and façades, are used to improve code reuse in Arcademis. Arcademis also uses design patterns to solve problems specific to the domain of distributed systems, such as the Acceptor-Connector pattern, which support different connection establishment policies [24].

The remaining of this paper is organized as follows. Section 2 describes the overall architecture of Arcademis as well as the main classes and design patterns used in its design. This section also documents the main aspects of the framework that can be independently configured. Section 3 presents the RME platform: a J2ME/CLDC remote method invocation system derived from Arcademis. RME also illustrates the flexibility provided by Arcademis, since traditional and monolithic middleware, such as Java RMI, are not available for the J2ME/CLDC platform. Section 4 gives an overview of existing configurable middleware systems and relates Arcademis with them. Section 5 presents concluding remarks.

2. Architecture

A distributed system built on top of Arcademis is structured on three abstraction levels. The first level comprises the framework components. Essentially these are abstract classes and interfaces, although Arcademis also provides concrete components that can be used without further extensions. The second level is represented by the concrete middleware platform obtained from Arcademis. The framework defers to this level decisions such as the communication protocol and the serialization strategy that will be adopted. Finally, the third abstraction level comprises components that provide services to end users. These components constitute what is normally called a distributed application.

Each instance of Arcademis has a central component called ORB. This element is implemented as a *singleton*, a design pattern that limits the number of instances of a given class to exactly one [7]. Besides being implemented as a singleton, the ORB contains a set of *object factories*. An object factory is a design pattern that is used to create instances of objects [7]. The main advantage of this pattern is to make it easier to change the implementation of a component without interfering in other modules of the system. For example, in Arcademis all communication channels are created by an object factory. In order to modify the transport protocol used by the middleware, for instance, from TCP to UDP, it is sufficient to change the channel factory bound to the ORB. Because channel factories must preserve the channel interface, the other components of the platform need not to be changed. Arcademis defines sixteen different factories, but its instances do not have to use all of them. Figure 1 outlines the general organization of middleware systems based on Arcademis.

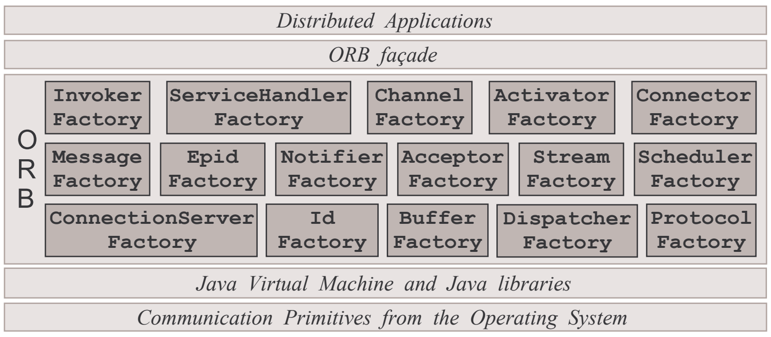


Figure 1. Factories defined by Arcademis.

Arcademis has been originally designed to support the implementation of object-oriented middleware platforms. According to this model, a client object uses intermediate components in order 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. It receives messages containing information about remote invocations and determines what method of the server should be executed. Although application developers have the illusion that the methods are processed locally, actually each remote call is transmitted by the stub to the skeleton and then to the implementation of the remote object. The results of remote invocations are transmitted across the opposite path.

Arcademis defines the basic structure of stubs and skeletons, but it does not predefine how stubs and skeletons should be generated. For example, middleware developers may decide to use a stub/skeleton compiler or may decide to generate stubs/skeletons dynamically using reflection. Nevertheless, in order to provide support for the automatic generation of stubs and skeletons, Arcademis includes the source code of `rmec`, a stub/skeleton compiler based on a tool available in the NinjaRMI system [9]. The application developer can reuse most of the classes and methods of this compiler in order to adapt the code that it generates to requirements posed by particular middleware systems.

Despite how they are generated, stub objects must extend `arcademis.Stub`, whereas skeletons are subclasses of `arcademis.server.Skeleton`. The stub implementation contains a reference to a factory of invokers (Section 2.9), and the address of the remote object that it represents (Section 2.6). The skeleton encapsulates the remote object. The `arcademis.server.RemoteObject` class, which must be implemented by each remote object, defines abstract methods for the creation of stubs and skeletons. Stubs are created by calling the lookup method of the discovery agency (Section 2.7); skeletons are normally created by the

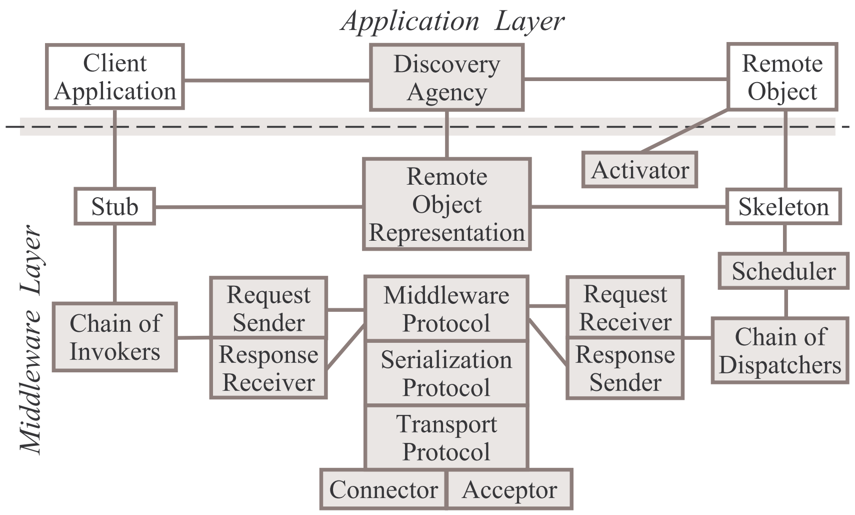


Figure 2. Main components of Arcademis.

activator component (Section 2.8) during the initialization of the remote object. The activator is also responsible for linking the skeleton, the dispatcher (Section 2.10), the request receiver (Section 2.2), and any other component that is part of the server architecture, such as the scheduler (Section 2.11).

Besides stubs and skeletons, Arcademis defines several other components that collaborate to define the middleware architecture and to support customizations. The most important of these elements are represented in Figure 2. 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 order remote calls according to their priorities. The network 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 provide to distributed applications. 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. These components are explained in detail in the remainder of this section.

There are eleven basic configurations that can be applied to middleware platforms derived from Arcademis. Although most configurations are orthogonal, some components of the framework can collaborate on two or more of them. The aspects that are subject to configurations in Arcademis are the following:

data transportation: comprises the techniques and protocols used in the transmission of raw sequences of bytes between nodes. Arcademis permits the configuration of different transport protocols and the addition of extra functions to communication channels (for example, encryption or compression of messages).

connection set up: defines how channels are established between nodes so that data can be sent across them. For example, it is possible to define synchronous or asynchronous connection establishment strategies or to add caches in order to reuse channels.

middleware protocol: defines the set of messages exchanged between distributed objects. This aspect supports the implementation of protocols customized to particular technologies, such as wireless networks. For example, in order to reduce the number of messages exchanged in wireless links, middleware designers may decide to eliminate distributed garbage collection messages [2].

serialization protocol: defines how the internal state of objects is converted into a raw sequence of bytes and vice-versa. For example, users of the framework can use a protocol based on the serialization package of Java or define their own protocol.

invocation semantics: determines the level of reliability provided by the implementation of remote invocations (e.g. best effort, at-most-once, at-least-once, etc).

remote object references: defines how remote objects are handled in distributed applications. For example, objects can be represented by a host name and object name pair or using a URL based notation.

service lookup: defines the mechanisms the middleware provides to discover distributed objects.

remote object activation: determines how a distributed object is made ready for receiving remote calls.

invocation protocol: defines how a remote call is processed at the client side. For example, users can customize this protocol in order to buffer, cache or log invocations.

dispatching protocol: defines how a remote call is processed at the server side. For example, users can customize this protocol in order to authenticate or log arriving calls.

priority policy: defines the order in which method invocations are delivered to remote objects. Priorities can be assigned, for example, to methods, or to clients.

The remainder of this section describes in more details the previously mentioned configuration aspects.

2.1. Data Transportation

In Arcademis, the transport protocol is implemented by two components: `Channel` and `ConnectionServer`. Channels are responsible for transmitting byte sequences between clients and servers, whereas the function of connection servers is to receive connection requests and to create channels. The framework does not assume the use of any specific transport protocol, and possible implementations can be based on UDP, TCP, HTTP, etc. In order to add further functionality to a channel, Arcademis uses the Decorator design pattern [7], which provides a way to modify the behavior of individual objects without creating new derived classes. A channel decorator is an object that implements the `Channel` interface and, in addition to this, has an attribute of the `Channel` type. As a subtype of `Channel`, the decorator can overwrite some of its methods in order to aggregate further capabilities to them.

Examples of capabilities that can be aggregated to channels by means of decorators include mechanisms for compressing or encrypting messages, check points or error correcting code for handling transmission failures, and buffers to improve performance or to allow undo operations. Figure 3 (a) shows an example of composition of decorators. `ZipChannel` compresses messages in order to make better use of the available bandwidth and `LogChannel` implements a report generator that yields a log file describing channel utilization. The `TcpSocketChannel` class is one of the concrete components provided by Arcademis. The same chain of capabilities could have been built using inheritance, but, in this case, it would not be so flexible. In Figure 3, nothing prevents `ZipChannel` from being inserted before the other decorator; moreover, a third decorator can be added to the sequence without requiring modifications in the implementation of the existing ones.

2.2. Connection Establishment

Connection set up has been implemented according to the *acceptor-connector* design pattern [24]. This pattern decouples the connection initialization from its processing, once the channel has been initialized. The main participants of the pattern are the *acceptor*, the *connector* and the *service handlers*, which are depicted in Figure 3 (b). The connector is responsible for contacting the acceptor when necessary to set up a channel between two hosts. Once the connection is established, the resulting channel is passed to a service handler, which sends and receives messages according to the application needs. One of the advantages of this pattern is the possibility of configuring different connection strategies without modifying the service handlers. Possible strategies include synchronous and asynchronous connection establishment and the use of caches in order to recycle channels.

2.3. Middleware Protocol

The middleware protocol is defined by a set of messages and by a state machine that determines how the messages are exchanged. In Arcademis, messages are marshalable implementations of the `Message` interface, and the sequence of bytes that composes it is given by the implementation of its `marshal` method (this method is further discussed in Section 2.4). Messages are implemented according to the Command design pattern [7]: each message

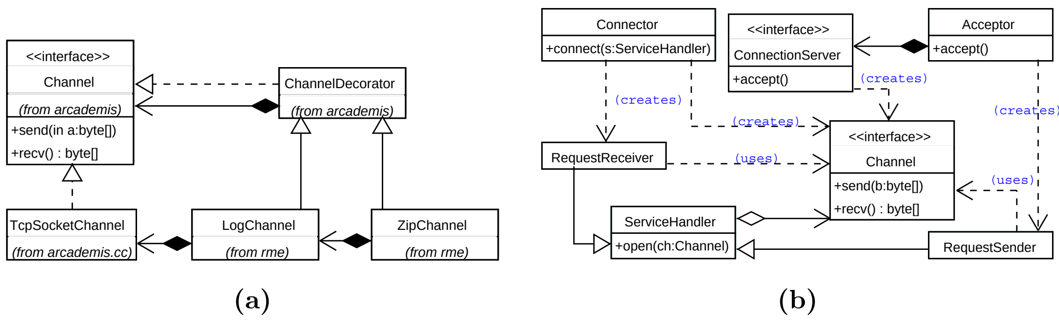


Figure 3. (a) Composition of decorators. (b) The acceptor-connector components.

implements a command that determines the actions to be executed after it is received. This approach makes it easier to modify the middleware protocol. Whenever a new message should be added to the system, it is sufficient to provide a new implementations for the `Message` interface. Because messages are typed structures, the same code can be used to handle all of them, by means of polymorphism and dynamic dispatching. The bridge between `Messages` and `Channels` is done by a component called `Protocol`. The function of this component is to marshal messages before sending them across channels and to unmarshal messages after a raw sequence of bytes is received.

2.4. Serialization Protocol

The serialization protocol used in Arcademis depends on serialization methods implemented by application developers. For this purpose, the framework defines the interfaces `Marshalable` and `Stream`. Serializable objects should implement the `Marshalable` interface, which declares two methods: `marshal` and `unmarshal`. The first method describes how an object is transformed into a sequence of bytes, whereas the second one defines how the state of the object can be recovered from such sequence. The `Stream` interface specifies the serialization protocol, i.e., a collection of methods for reading and writing sequences of bytes. An example of class that implements `Marshalable` is presented in Figure 4.

Since the serialization protocol is independent from the built-in serialization package of Java, middleware systems derived from Arcademis can be used for example in J2ME/CLDC, where the default serialization package is not available.

2.5. Invocation Semantics

The invocation semantics defines the reliability level that the middleware implementation provides to application developers. In Arcademis, this semantics is determined by the implementation of four components, which are interposed between stubs and skeletons: the *request-sender* and the *request-receiver* at the client side, and the *response-sender* and the

```
import arcademis.*;

public class Person
implements Marshalable {
  private String name = null;
  private int age = 0;
  private boolean isMan = false;

  public void marshal(Stream b)
  throws MarshalException {
    b.write(name);
    b.write(age);
    b.write(isMan);
  }

  public void unmarshal(Stream b)
  throws MarshalException
    name = (String)b.readObject();
    age = b.readInt();
    isMan = b.readBoolean();
  }

  // implementation of the other methods
}
```

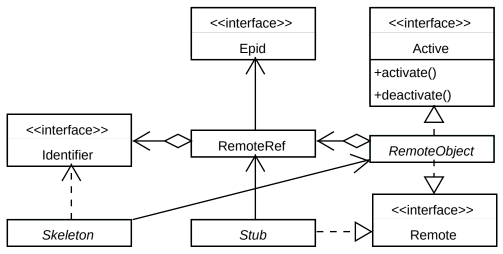
Figure 4. Example of serializable class.

response-receiver at the server side, as shown in Figure 2. These components are service handlers, as described in Section 2.2. The three most popular invocation semantics used in object oriented middleware are *best-effort*, *at-most-once* and *at-least-once* [4]. The first of them does not provide any guarantee regarding the processing of remote calls. In the presence of failures, they may be executed once, several times or even may not be executed. The second one assures that remote invocations will be processed only once or will not be executed. Finally, the at-least-once semantics gives the client application the guarantee that remote calls will be executed at least one time.

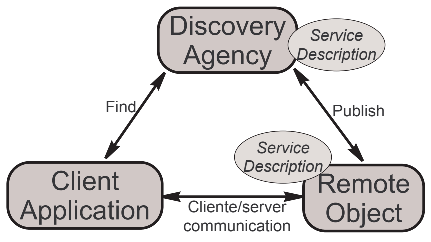
2.6. Remote Object References

In Arcademis, distributed objects are handled using remote references, which are instances of subclasses of the `RemoteReference` abstract class. The implementation of this component determines how a distributed object is distinguished from others and the semantics of operations such as `equals` and `toString` when invoked remotely. The identifier and address of a remote object are represented by the interfaces `Identifier` and `EndPointIdentifier`, respectively. These components can be implemented in different ways. For instance, in CORBA, remote objects are identified using Interoperable Object References (IOR). Among other information, IORs contain the host and port address of the target object. In SOAP, as common in the Web, remote objects are identified by means of Uniform Resource Identifiers (URI).

Distributed objects have to inherit from the `RemoteObject` class. In addition, remote objects must implement the `Remote` interface. Although this interface is empty, i.e., it does not declare any method, it is used by the system to distinguish references to local objects from references to remote objects. For example, in remote invocations, the implementation of Arcademis should replace remote references by their associated stubs, in order to simulate call by reference. The relations among the components described in this section are depicted in Figure 5 (a).



(a)



(b)

Figure 5. (a) Components for remote object representation. (b) Service-oriented architectures.

2.7. Service Lookup

Middleware platforms derived from Arcademis can be described as service-oriented architectures [3]. Such architectures have three different actors: service providers, service requesters and discovery agencies. Service providers are represented by remote objects, whereas requesters are represented by clients in general. The discovery agency, or lookup service, is an independent element that should be provided by all instances of Arcademis. It permits clients to find and gain access to remote objects. The three main actors of service-oriented architectures are depicted in Figure 5 (b).

Client and Server applications access the discovery agency by means of different interfaces. Service providers register themselves using a `publish` operation, while clients use a `find` operation to look for distributed objects. Objects are registered in the discovery agency using a name (a string) or the interface they implement. Other forms of representation can be provided by middleware designers.

2.8. Remote Object Activation

The activation of remote objects in Arcademis is implemented by a component called `Activator`, which allocates the resources the server needs to process remote invocations. For example, it initializes data structures internal to the middleware and creates threads to wait for remote calls. It also generates a proper identifier to the remote object and binds it to an unused network address. The `RemoteObject` class implements the `activate` and `deactivate` methods, which are used to interact with the activator. Depending on the activation policy adopted, the `activate` method may have to be invoked explicitly by application developers or it may be called automatically during the instantiation of remote objects.

2.9. Invocation Protocol

In Arcademis, remote methods are invoked by a component called `Invoker`. The main functions of invokers are: (i) to create a connection with the server or to reuse one if possible; (ii) to create messages containing the arguments of remote calls; (iii) to create service handlers to send calls and to wait for their results. Invokers can also be customized in order to reuse connections across successive calls or to create a new connection whenever a method invocation is requested.

In order to aggregate further responsibilities to a invoker, Arcademis provides an *invoker decorator*, which is used in the same way as the channel decorator described in Section 2.1. Examples of capabilities that may be aggregated to invokers are: caches (to avoid the transmission of already requested calls), buffers (to group several remote calls together in order to make better use of the available bandwidth) and log generators. It is also possible to use invoker decorators to implement asynchronous calls. In this type of call, a separate *thread* is created to process each remote invocation, so that clients do not remain blocked during remote processing. In this case, results of remote invocations are inserted into a buffer that clients can inspect afterwards.

2.10. Dispatching Protocol

The `Dispatcher` is the component responsible for transmitting to a remote object the requests directed to it. The implementation of the dispatcher determines the general structure of middleware server side. Figure 6 presents an example of server organization. In this example, there are three active objects: the activator, the scheduler and the response sender. Call descriptors (`rc`) are inserted into a queue and ordered by the scheduler, before being sent to the remote object. Results of remote invocations are inserted into another queue, and are asynchronously transmitted to clients by the response sender.

In addition to channel and invoker decorators, Arcademis also supports dispatcher decorators. Examples of capabilities that can be added to dispatchers by means of decorators include the implementation of security policies, the generation of log files describing server usage, the report of the server load rate to clients, the redirection of calls to other servers and the creation of threads in order to process specific calls.

2.11. Priority Policy

Arcademis supports the definition of priorities among remote calls. The `Scheduler` is the component of the framework in charge of applying such priorities. Three possible priority policies, from the simplest to the most complex, are: the assignment of priorities to remote methods, the assignment of priorities to clients and the assignment of priorities to endpoints in the server. In the last case, it is assumed that servers may receive requests in more than one endpoint. Besides changing the scheduler, the implementation of priorities may also require changes in other components. For example, in order to assign each remote method a different priority, it is necessary to modify the implementation of stubs.

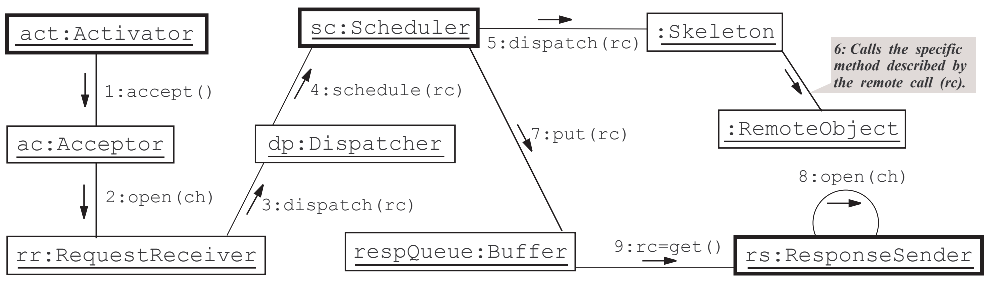


Figure 6. Representation of the main components of Arcademis.

2.12. Implementation

The current implementation of Arcademis has about 4250 lines of code, including 34 interfaces, 11 abstract classes and 36 concrete classes. The system is non-monolithic in the sense that the set of components that should be present in its instances has being kept as small as possible. Some required elements are common both to the client and server sides of derived middleware systems; for instance, communication channels, remote object references, wire protocol, remote address descriptors, byte streams etc. Other components are exclusive of the client side. This set includes the connector, the invoker, the request sender and the stub. The minimum structure of the server side of an Arcademis instance is more complex. Among the mandatory elements are the activator, the acceptor, the connection server, the dispatcher, the skeleton, the request response, and the remote object itself.

Most of Arcademis size is due to optional components. These elements may not be present in every instance of Arcademis. Examples of optional components are the scheduler, the response sender, the response receiver, invoker decorators, dispatcher decorators, channel decorators, event handlers, and event notifiers. The realization of Arcademis is defined at compile time, by the specification of the object factories that will be used.

3. RME: RMI for J2ME/CLDC

In order to validate Arcademis, the framework was used to derive a remote invocation service for Java 2 Micro Edition, a Java distribution targeting resource constrained devices such as cell phones and palmtops [18]. The J2ME platform is divided into configurations. Each configuration comprises a Java Virtual Machine and a minimal set of libraries that should be available for the group of devices that meet the minimum hardware requirements defined by the configuration. Presently, J2ME provides two main configurations: CDC (*Connected Device Configuration*) and CLDC (*Connected, Limited Device Configuration*). CDC groups less-restrictive devices that can afford at least 2MB of memory and persistent network connections,

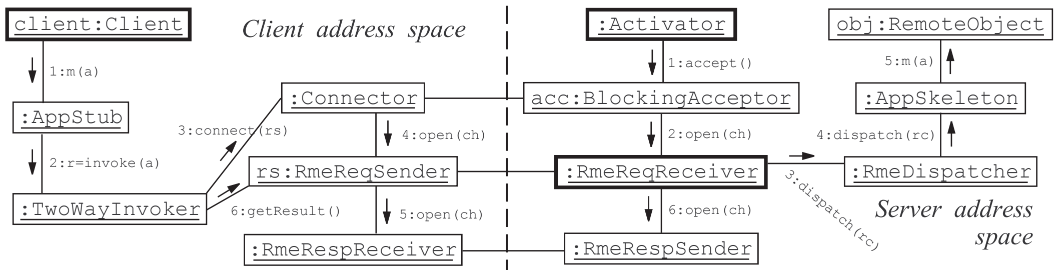


Figure 7. Architecture of RME.

like set-top boxes. This configuration provides the fundamental APIs from J2SE, including support for reflection. The CLDC configuration is suitable to more resource constrained devices, generally mobile and battery-operated, with low bandwidth and intermittent network connections, like cell phones and low-end PDAs. CLDC defines a subset of the J2SE API which does not include reflection. Since object serialization in Java RMI depends on reflection, this middleware is not available for the CLDC configuration.

RME (*RMI for J2ME/CLDC*) is an instance of Arcademis that provides to the CLDC configuration of J2ME a remote method invocation service. RME provides distinct services to the client and server sides of a distributed application. Only the client part of the middleware is implemented in J2ME. Its server counterpart is implemented in J2SE. This is not a limitation of Arcademis architecture but merely an implementation decision targeting devices with constrained resources (at the cost of not allowing clients to export remote objects).

The main components involved in the execution of a remote call in RME are described in Figure 7. RME is a synchronous service, meaning that client threads remain blocked during the execution of remote calls. In the server side, the activator and the request receivers are active objects, that is, they execute in a separated thread, and a new thread is created for each incoming connection. This arrangement permits to separate the thread in which connections are received (the acceptor thread), from the threads in which connections are handled (the request receiver thread). In this figure, `AppStub` and `AppSkeleton` are, respectively, stubs and skeletons that were generated by a tool called `rmec`. This tool is a version of the stub/skeleton compiler that is part of the Arcademis framework (Section 2).

The basic invoker provided by RME (Section 2.9) is called `TwoWayInvoker`. Its implementation performs remote calls in a two-way basis, that is, the client application is expected to receive an answer for each remote call issued. `Rmec` assigns to each remote method a basic invoker implementation, possibly augmented by a sequence of decorators. In Figure 7, for example, `rmec` has assigned to method `m()` an instance of `TwoWayInvoker` without any decorator. In addition to the basic invoker implementation, RME provides three pre-defined invoker decorators:

- `Cache`: caches may be used to store the result of a remote invocation in an attempt to reuse it later if the same method is triggered again. A cache is particularly useful when the associated method does not generate side-effects.
- `Timer`: this decorator allows to define time limits for the execution of remote calls. If a remote call is not performed in the specified amount of time, an exception is raised.
- `Active`: this decorator creates a separate thread to carry out remote method invocations, so that the main application does not remain blocked during the processing of the call.

RME uses TCP/IP for data transmission. The middleware protocol adopted by RME is named RMEP (RME Protocol), and it defines six different types of messages: *call*, *return*, *ping*, *ack*, *inq* and *load*. The *Call* message describes a remote invocation, including its arguments and identifiers. *Return* messages holds the results of remote calls. *Pings* and *acks* are mostly used in order to verify if servers or clients are alive. The *inq* message is used by clients in order to discover the load on specific servers, which is informed by means of a *load* message.

Two different invocation semantics are available in RME: best-effort and at-most-once. The adoption of one of them is just a matter of assigning to the ORB the proper service handler factory. Figure 8 shows the state machine that describes the implementation of the at-most-once semantics. According to that scheme, the request sender, after issuing a call, starts a time counter, and if a certain interval has elapsed before it receives any response, it repeats the process. After performing a number of calls without answer, the client application assumes that the server is not achievable and an exception is raised. The request receiver, on the other hand, keeps identifiers of received calls on a list, in order to discard repeated requests. In this list, identifiers of remote calls are associated to their return values, and if an already processed request is received, the server answer it without further processing.

Best-effort semantics does not guarantee the execution of a remote call, although it reports any failure to the client thread. The request sender, after issuing the call, remains blocked waiting for the return value or a timeout. If the remote invocation cannot be accomplished in the expected amount of time, an exception is thrown, and no further processing is performed. The request receiver simply answers remote invocations as they come.

RME gives to application developers a programming syntax similar to the one provided by Java RMI. Remote methods must be declared in an interface that extends the `arcademis.Remote` interface and must declare the possibility of throwing `arcademis.ArcademisException`. Remote object classes must implement this interface and extend the `RmeRemoteObject` class. Figure 9 (a) shows an example of remote interface and Figure 9 (b) depicts its implementation. In the given example, the remote method simply sums two integer numbers. Figures 9 (c) and 9 (d) show, respectively, examples of server and client code. The server creates a remote object, and the client invokes a remote method on it. Distributed applications based on RME should configure the ORB before starting its execution, using an instance of the `RmeConfigurator` class. The `configure` operation determines the factories associated with the ORB. The discovery agency of RME is implemented by the `RmeNaming` class, and it defines the same set of methods provided by the class `java.rmi.Naming`. Because RME targets resource constrained devices, `RmeNaming` creates stubs according to the *Flyweight* design pattern [7]: before generating a stub, the discovery agency checks whether there is an stub instance already bound to the same remote object. If there is, a reference to this instance

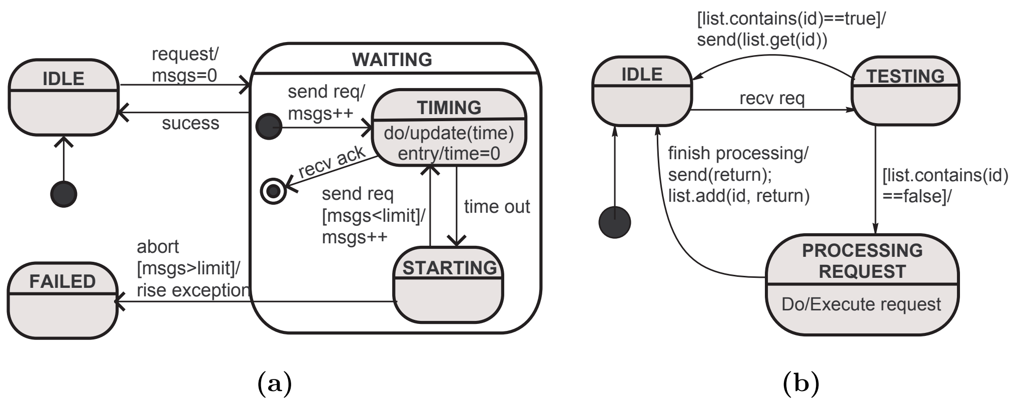


Figure 8. At-most-once call semantics: (a) request-sender state machine. (b) request-receiver state machine.

is returned.

Applications Examples of applications based on Arcademis/RME can be found in the documentation section of the Arcademis web page[†]. The remote invocation capabilities provided by RME permit application developers to improve the processing and the storage capacity of mobile devices. For instance, one of the applications that we have implemented using RME is a phone book. The phone information is made available in a database in the fixed network, which can be accessed by means of remote calls. The amount of persistent memory available in a typical cell device makes it prohibitive to store locally a large database. On the other hand, the programming syntax provided by RME conceals the underlying network, allowing software developers to focus their attention in the application logic.

In another application, which coordinates a stock market system, clients (buyers) and service providers (vendors) interact by means of a central server. Clients can issue quests to a specific vendor, in order to wait for events. For instance, an event can be triggered when the price of the share of a certain company falls under a given threshold. The cell phone application does not support call backs; however, the use of the *active* and *timer* decorators permit the client to open a connection in order to wait for event notifications, or a timeout, without being blocked.

[†]http://www.dcc.ufmg.br/lpp/arcademis

```
import arcademis.*;
public interface RemInt
extends Remote {
  public int sum(int a, int b)
  throws ArcademisException;
}
```

(a)

```
import rme.*;
import rme.server.*;
public class RemObj extends
RmeRemoteObject implements RemInt {
  public int sum(int a, int b) {
    return a + b;
  }
}
```

(b)

```
import rme.*;
import rme.naming.*;
public class Server {
  public static void main(String a[])
  throws Exception {
    RmeConfigurator c =
    new RmeConfigurator();
    c.configure();
    RemObj o = new RemObj();
    RmeNaming.bind("obj", o);
    o.activate();
  }
}
```

(c)

```
import rme.*;
import rme.naming.*;
public class Client {
  public static void main(String a[])
  throws Exception {
    RmeConfigurator c =
    new RmeConfigurator();
    c.configure();
    RemInt i=(RemInt)
    RmeNaming.lookup("obj");
    i.sum(2, 2);
  }
}
```

(d)

Figure 9. (a)Remote Interface. (b)Remote Object. (c)Server. (d)Client.

4. Related Work

Conventional object-oriented middleware, such as CORBA [16], Java RMI [27] and .NET [15], have been designed in order to fulfill the requirements of a wide range of applications and technologies. Despite this, several versions of these systems were later proposed in order to address particular application domains. For example, there are at least the following versions of CORBA: minimum CORBA, Real-Time CORBA, Fault-Tolerance CORBA and High-Performance CORBA. This proliferation of versions of the same middleware technology justifies the implementation of frameworks that capture the fundamental architecture of a given middleware system and that promote its reuse in the various flavors of the system. Conventional middleware systems have also few hooks for customizations demanded by application developers. For this purpose, CORBA supports the concept of interceptors, and Java RMI supports the installation of stub factories. Thus, middleware users should also benefit from design patterns that add extensibility and flexibility to middleware architectures.

Non-monolithic middleware systems have been the focus of intense research since the mid-1990s. Configurations can be carried out statically (at compilation time) or dynamically (at execution time). One example of statically configured middleware is the TAO platform [23]. This system, which has been designed to support real time applications, presents a modular architecture based on design patterns. Some of these patterns, such as *acceptor-connector* [24], have been used in the implementation of Arcademis. Among the examples of dynamically reconfigurable middleware, it is possible to cite dynamicTAO [13], UIC CORBA [20], Open ORB [1] and OpenCORBA [14]. Dynamic configuration can be accomplished in a number of ways; however, the use of computational reflection seems to be the most accepted trend. The essential difference between Arcademis and the previous platforms is the fact that Arcademis is not a middleware targeting a specific application domain; it is a framework from which middleware can be developed. In addition, the architecture of Arcademis presents some unique characteristics. For instance, concepts such as chains of invokers and dispatchers, channel decorators, flyweight stubs, and the command pattern in the implementation of messages are not observed in the former platforms.

Another framework with similar objectives is Quarterware [25]. This system has been used in the development of platforms compatible with CORBA, Java RMI and MPI. The main difference between Arcademis and Quarterware concerns the aspects that are subject to configuration in each framework. Quarterware supports six classes of configuration, whereas Arcademis supports eleven. Some of the customizations possible in Arcademis are missing in Quarterware or have been merged into a single aspect. For example, the dispatching strategy of Quarterware, that comprises remote object discovery and data transmission, is separated in Arcademis into three different aspects: service discovery, invocation protocol and dispatching policy.

.NET Remoting is a communication framework targeting the .NET platform [15]. The system supports redefinition of several low-level communication components, including channels, formatters and remote references. However, the system lacks support to high-level middleware reconfiguration aspects, including the middleware protocol, connection set up strategy and invocation semantics.

Dynamically Programmable and Reconfigurable Software (DPRS) is a technique to support the construction of dynamically configurable, updateable, and upgradeable middleware services [19]. DPRS systems are based on the three abstractions: micro building blocks (MBB), actions and domains. A MBB represents the smallest function unit in the system, an action coordinates the execution of MBBs, and a domain is a collection of related MBBs. ExORB is a Java prototype middleware based on DPRS abstractions. The system is composed of 28 micro building blocks that support client only or client/server functionality, and IIOP or XMLRPC middleware protocols. The MBB-based architecture of ExORB supports on-the-fly updates and upgrades. In Arcademis, updates and upgrades are always defined at compile time, creating new object factories and adding decorators to existing components. On the other hand, the ExORB performance ranges from 45% to 75% of the performance of an equivalent static implementation of the system. Certainly, this overhead will increase if the logic and the structure of ExORB becomes more complex, including MBBs and domains to support interceptors, invokers, dispatchers, activators, schedulers, service handlers etc.

Another research thread related to Arcademis includes mobile computing middleware platforms, such as UIC CORBA [20] and LegORB [21]. Similar to RME, these platforms provide a kernel in which only the essential components of a middleware system are presented. There is also an implementation of Java RMI, the RMI Optional Package (RMI OP) [10], that targets the CDC configuration of J2ME.

5. Conclusion

This paper has presented Arcademis, a Java-based framework for communication middleware development, and RME, a J2ME/CLDC based middleware derived from Arcademis. Arcademis is a flexible and non-monolithic communication middleware framework. Arcademis is flexible because middleware systems derived from it are ultimately defined by a set of factories associated with the ORB object. Thus, it is possible to modify the default behavior of the middleware by just changing the factories that create the components responsible for this behavior. For instance, in RME the call semantics, the transport protocol and the invocation strategy were modified in this way. Moreover, a J2SE implementation of RME can be accomplished by changing just two factories of the system: the `Channel` and `ConnectionServer` factories.

The factory-based design has also contributed for making Arcademis a non-monolithic framework. Unnecessary components will not be used if their factories are not associated with the ORB. For example, in RME all remote methods always have the same priority; therefore, the `Scheduler` component is not necessary, and its corresponding factory is not bound to the ORB. Moreover, the client version of RME runs on J2ME and the server version on J2SE. The former uses 10 factories, the latter uses 13, and the full framework defines 16 factories. As a consequence, Arcademis can be used in the instantiation of middleware systems that accommodate just the requirements presented by a particular application.

Besides RME, Arcademis has already been used in the implementation of a middleware system that supports the annotation of remote interfaces with tactics that can deal with phenomena typical of distributed settings [17]. Tactics are described in a high level language, and can be used to assign priorities to remote methods, to benefit from the existence of several service providers, to define the reliability level of remote calls and to associate decorators to remote methods.

Arcademis and RME are available from `http://www.dcc.ufmg.br/llp/arcademis`.

REFERENCES

1. Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fábio M. Costa, Hector A. Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui S. Moreira, Nikos Parlavantzas, and Katia B. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.
2. Stefano Campadello, Oskari Koskimies, and Kimmo Raatikainen. Wireless Java RMI. In *4th International Enterprise Distributed Object Computing Conference*, pages 114 – 123. IEEE Computer Society, 2000.
3. Michael Champion, Chris Ferris, Eric Newcomer, and David Orchard. *Web Services Architecture*. W3C, 2002.
4. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*, volume 1. Addison-Wesley, 2nd edition, 1996.
5. Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109 – 126, 2002.
6. Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *22nd International Conference on Software Engineering - Future of Software Engineering Track*, pages 117 – 129. ACM Press, 2000.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. Kurt Geihs. Middleware challenges ahead. *IEEE Computer*, 34:24 – 31, 2001.

9. Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, B. Zhao, and Robert C. Holte. The Ninja architecture for robust internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.
10. Sun Microsystens Inc. RMI optional package specification version 1.0, 2003. http://java.sun.com/products/rmiop/ – last visit: June 2004.
11. Ralph E. Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10 – 17, 1997.
12. Fábio Kon, Fábio Costa, Roy Campbell, and Gordon Blair. The case for reflective middleware. *Communications of the ACM*, 45(6):33 – 38, 2002.
13. Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhes, and Roy Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 2000.
14. Thomas Ledoux. OpenCorba: a reflective open broker. In *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 197 – 214. Springer-Verlag, 1999.
15. Piet Obermeyer and Jonathan Hawkins. Microsoft .NET Remoting: A technical overview. Technical report, Microsoft Corporation, 2001.
16. OMG. CORBA IIOP 2.3.1 Specification. Technical Report 99-10-07, Object Management Group, 1999.
17. Fernando Magno Quintao Pereira, Marco Tulio de Oliveira Valente, Wagner Salazar Pires, Roberto da Silva Bigonha, and Mariza Andrade da Silva Bigonha. Tactics for remote method invocation. *Journal of Universal Computer Science*, 10(10):824–846, 2004.
18. Roger Riggs, Antero Taivalsaari, and Mark VandenBrink. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. Addison Wesley, 2001.
19. Manuel Roman and Nayeem Islam. Dynamically programmable and reconfigurable middleware services. In *5th ACM/IFIP/USENIX International Conference on Middleware*, pages 372–396. Springer Verlag, 2004.
20. Manuel Román, Fábio Kon, and Roy Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001. Electronic Edition.
21. Manuel Román, Dennis Mickunas, Fábio Kon, and Roy Campbell. LegORB and ubiquitous CORBA. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, pages 1 – 2. Springer-Verlag, 2000.
22. Richard E. Schantz and Douglas C. Schmidt. *Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications*. Wiley & Sons, 2001.
23. Doublas Schmidt and Chris Cleeland. Applying patterns to develop extensible and maintainable ORB middleware. *Communications of the ACM*, 40(12), 1997.
24. Douglas Schmidt. Acceptor-Connector – an object creational pattern for connecting and initializing communication services. In *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, chapter 3. Wiley & Sons, 2000.
25. Ashish Singhai, Aamod Sane, and Roy H. Campbell. Quarterware for middleware. In *IEEE International Conference on Distributed Computing Systems*, pages 192 – 201, 1998.
26. Sun Microsystems. Java Remote Method Invocation specification, revision 1.8, 2003.
27. 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 Association, 1996.