Arcademis: A Java-Based Framework for Middleware Development

Fernando Magno Quintão Pereira[1]*
Marco Túlio de Oliveira Valente[2]
Roberto da Silva Bigonha[1]
Mariza Andrade da Silva Bigonha[1]

[1] Department of Computer Science,
Federal University of Minas Gerais

[2]Department of Computer Science,
Pontifical Catholic University of Minas Gerais

{fernandm, bigonha, mariza}@dcc.ufmg.br,mtov@pucminas.br

Abstract. *This paper presents Arcademis, a Java-based framework for middleware development. Arcademis consists of a set of abstract classes and interfaces that define the general architecture of middleware systems. The main objective of Arcademis is to support the implementation of non-monolithic and easily reconfigurable middleware. In order to illustrate the use of the framework, the paper also describes the RME system. RME is a middleware derived from Arcademis that adds a remote method invocation service to distributed applications built on the CLDC configuration of Java 2 Micro Edition (J2ME).*

1. Introduction

In the last ten years, distributed systems developers have often relied on middleware to increase their productivity. Residing between the operating system and distributed applications, these platforms provide high-level 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. There are different kinds of middleware, such as message passing systems, tuple-space based systems and object oriented systems. However, object-oriented middleware – such as CORBA [OMG, 1999] and Java RMI [Sun, 2003] – are the most popular ones at the present time. In such middleware, developers can invoke methods on remote objects using a syntax similar to local invocations. In this way, interactions between local and remote processes give the impression of coexisting in the same address space.

Object oriented middleware have always been designed to make location transparent to developers of traditional distributed systems, i.e., systems running in personal computers connected by local or corporate networks. However, in recent years, the distributed environment has faced many changes. Nowadays, there are several kinds of computing

*Supported by CNPq and FAPEMIG.

devices (sensors, cell phones, PDAs, multicomputers, clusters, etc), several network infrastructures (Internet, wireless networks, grids, etc), several transport protocols (TCP, HTTP, etc) and applications with different quality of service requirements (real time systems, multimedia, mobile systems, electronic commerce, etc). On the other hand, conventional object oriented middleware are monolithic and inflexible systems, which can not be easily reconfigured to meet the requirements of rapidly changing technologies.

In order to address the limitations of current middleware implementations, this paper presents Arcademis[1]: 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 or technology. For example, this framework has been used to provide a remote method invocation system for J2ME/CLDC, the Java technology that targets mobile devices with limited computing resources, such as cell phones and palmtops. Middleware derived from 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 the platforms derived from Arcademis.

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 the programmer [Johnson, 1997]. Design patterns document recurring solutions to problems in software development [Gamma et al., 1994]. In Arcademis, frameworks and design patterns are applied synergistically to promote the implementation of flexible and non-monolithic middleware. As a framework, Arcademis predefines the overall architecture of a middleware system, so that developers can concentrate on the details of their particular applications. Moreover, well-known design patterns, such as Singleton, Abstract Factory, Strategy, Decorator and Façade, are used to increase Arcademis flexibility. The framework also uses design patterns to face problems specific to the distributed system domain, such as patterns that support different connection establishment policies (Acceptor-Connect pattern) [Schmidt, 1997] and invocation semantics (Request-Response pattern).

The remaining of this paper is organized as follows: Section 2 gives an overview of existing reconfigurable middleware systems and compares Arcademis with some of them. In Section 3, the overall architecture of Arcademis is presented, and the main classes and design patterns used in this framework are described. This section also documents the aspects of the framework that can be specialized and reconfigured. Section 4 presents the RME platform: a J2ME/CLDC remote method invocation system derived from Arcademis. RME illustrates the flexibility provided by Arcademis, since traditional and monolithic Java middleware, like Java RMI, are not available in the J2ME/CLDC platform. Finally, Section 5 presents concluding remarks.

2. Related Work

Research related to non-monolithic middleware systems started in the end of the last decade. Examples of such platforms are TAO, dynamicTAO and UIC CORBA.

[1]In Portuguese, Arcademis is a coined word from the initials of framework for middleware development

TAO [Schmidt and Cleeland, 1999] targets real time applications, and its architecture is strongly based on design patterns. Some of these patterns, such as the *acceptor-connector* [Schmidt, 1997], have been employed in the Arcademis implementation. DynamicTAO [Román et al., 2001] adds dynamic reconfiguration to TAO, which only can be customized statically, that is, at compilation time. UIC CORBA [Román et al., 2001] is also a dynamically reconfigurable middleware that, similar to RME, targets mobile devices. DynamicTAO and UIC CORBA are examples of reflective middleware, that is to say, these platforms provide reconfiguration by means of reflection: a mechanism that allows a program to know aspects of its internal structure during execution time. In order to keep its core simple, Arcademis, like TAO, does not provides mechanisms for implementing dynamic reconfigurations, although some customizations can be performed by the application developer during execution time, as discussed in Section 3.9.

Arcademis is not a middleware platform, as TAO, dynamicTAO and UIC. It is a framework that allows the derivation of middleware systems. Another framework with similar objectives is Quarterware [Singhai, 1999]. This framework has been used in the development of systems compatible with CORBA, Java RMI and MPI [Singhai, 1999], a message oriented middleware. The main difference between Arcademis and Quarterware is related to the configuration parameters defined by each framework. In Arcademis, some of the configurable aspects outlined by Quarterware have been divided into two or more different parameters, in order to provide developers with greater flexibility when necessary to configure middleware. For example, the dispatching strategy of Quarterware, that comprises remote object discovery and data transmission has been separated into three different parts: service discovery, invocation policy and dispatching policy. Therefore, while Quarterware defines six parameters for configuration, Arcademis determines eleven, as described in Section 3.

Arcademis is implemented in Java, and its instances target devices able to execute a Java Virtual Machine. There are several examples of middleware systems implemented in Java, such as Java RMI or JacORB [Brose, 1997], and there is an implementation of Java RMI targeting CDC (Connected Device Configuration), another configuration provided by the J2ME platform. However, there is a lack of java-based frameworks for middleware development, and the most traditional platforms do not present too many options for customization. Java RMI itself provides few opportunities for configuration: this platform supports a predefined marshalling and unmarshalling algorithm (based on reflection), only one invocation semantics (synchronous, with at-most-once reliability level) and only one thread policy (a new thread per connection) [Sun, 2003].

3. Architecture of Arcademis

A distributed system built on top of Arcademis is structured on three abstraction levels. The first of these levels is composed of 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 as an instance of Arcademis. The framework defers to this level decisions such as the communication protocol and the serialization strategy that will be adopted. Finally, the third programming level comprises all the 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 maximum number of instances of a given class to exactly one [Gamma et al., 1994]. The ORB can also be characterized as a set of *Object Factories*. An Object Factory is another design pattern that is used to create instances of objects. The main advantage of this pattern is to make it easier to change a component's implementation 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 the factory preserves the channel interface, the other components of the platform need not to be changed.

Although several different types of middleware systems can be derived from Arcademis, this framework has been originally devised 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, that exist on the client side of a distributed application, and the skeleton, that 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 all the 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 being locally processed, 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.

Besides stubs and skeletons, Arcademis defines several other components that collaborate to outline the middleware architecture and to support customizations. The most important of these elements are represented in Figure 1. 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, serialization protocol and middleware protocol. Connections are established by two components: the `Connector` and the `acceptor`. Request *senders* and *receivers* determine the reliability level the middleware provide to distributed applications. Finally, the `Activator` determines how an object is made ready for receiving remote calls. Each of these components are better explained 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:

Transport Protocol: comprises the techniques and protocols used in the transmission of raw sequences of bytes between nodes;

connection set up: defines how channels are established between nodes so that data can be sent across them.

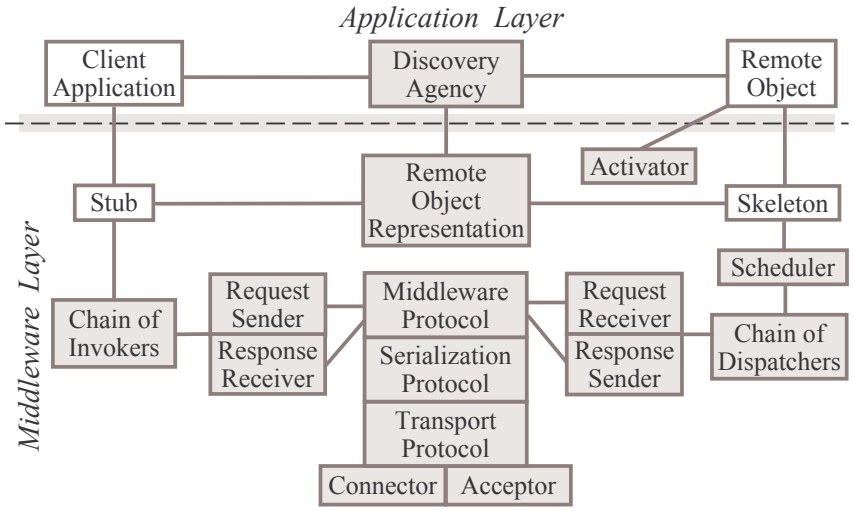


Figure 1: Representation of the main components of Arcademis.

middleware protocol: defines the set of messages exchanged between distributed objects;

serialization policy: defines how the internal state of objects can be converted into a raw sequence of bytes and vice-versa.

call semantics: determines the level of reliability provided by the implementation of remote calls (i.e. best effort, at most once, at least once, etc).

remote object representation: defines how remote objects are represented in distributed systems;

service lookup: defines the mechanisms the middleware provides to application developers so that distributed objects can be discovered;

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

invocation policy: defines how a remote call is invoked, that is, how it is converted into a byte sequence and sent across a channel.

dispatching policy: determines how a remote invocation is delivered to the skeleton once it has been retrieved from the transport network.

priority policy: defines the order in which method invocations are delivered to the actual implementation of the remote object;

The remainder of this section describes in details the customization possibilities provided by the previously mentioned configurable parameters.

3.1. Transport Protocol

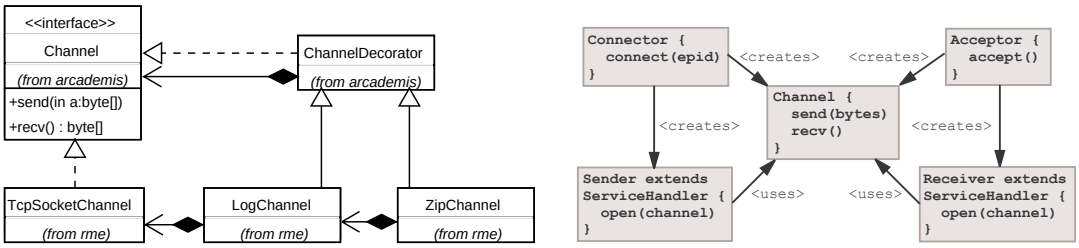
In Arcademis, the transport protocol is implemented by two components: `Channel` and `ConnectionServer`. Channels are responsible for transmitting byte sequences between clients and servers, whereas the function of connection servers is to receive connection requests and to create channels. The framework 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 [Gamma et al., 1994], 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 extra 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 2 (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 being built by means of inheritance, but, in this case, it would not be so flexible. In Figure 2, nothing prevents `ZipChannel` from being inserted before the other decorator; moreover, a third decorator can be added to that sequence without the need of modifying the implementation of the existing ones. Simple inheritance does not afford such flexibility.

3.2. Connection Establishment

Connection set up has been implemented according to the *acceptor-connector* design pattern [Schmidt, 1997]. 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 2 (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 is used to send and receive messages according to the distributed application needs. One of the advantages of this design pattern is the possibility of configuring different connection strategies without the need of modifying the service handlers code. Possible strategies include synchronous and asynchronous connection establishment and the use of caches in order to reuse channels.



(a) **(b)**

Figure 2: (a) Composition of decorators. (b) The acceptor-connector components.

3.3. Middleware Protocol

The middleware protocol is defined by a set of messages and by a state machine that determines how messages are exchanged in the system. 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 3.4). Messages are implemented according to the *Command* design pattern [Gamma et al., 1994]: each message 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 message objects 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.

3.4. Serialization Strategy

The serialization policy 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 3.

```
import arcademis.*;

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

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

```
  // implementation of the other methods

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

Figure 3: Example of serializable class.

3.5. Call Semantics

Skeletons and stubs communicate by means of four different service handlers that constitute a design pattern, proposed on this research, called *request-response* [Pereira, 2003]. These service handlers are called *request-sender*, *request-receiver*, *response-sender* and *response-receiver*, as described in Figure 1. The major advantage of this pattern is the possibility of easily reconfiguring the semantics of remote calls. The three most popular invocation semantics used in object oriented middleware are *best-effort*, *at-most-once* and *at-least-once*. 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 semantics known as at-most-once assures that remote invocations will be processed only once or will not be executed. Finally, the at-least-once reliability level gives the client application the guarantee that remote calls will be executed at least one time.

3.6. Remote Object Representation

In Arcademis, distributed objects are handled using remote references, which are implemented by the `RemoteReference` class. By modifying the implementation of this component, it is possible to configure how a distributed object is distinguished from others and the semantics presented by operations such as `equals` and `toString` when invoked remotely. The identifier and address of a remote object is implemented by the classes `Identifier` and `EndPointIdentifier`, respectively. These components can be implemented in different ways. For instance, in CORBA, remote addresses are defined as a pair formed by a host name and a port number; in SOAP, an object can be identified by the host address, an optional port number and a file system path. Identifiers can also be implemented in a number of ways. When not necessary to discriminate a really large number of elements, they can be defined as single integer numbers. On the other hand, in more scalable systems the identifier implementation should grant with high probability that in the distributed network there will not be two distinct remote objects holding equal identifiers.

Distributed objects have to inherit from the `RemoteObject` class, that determines the semantics of operations such as `equals` and `hashcode` when locally invoked. 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 Arcademis implementation 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 4 (a).

3.7. Service Lookup

Middleware platforms derived from Arcademis can be described as service-oriented architectures. 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 name service, is an independent element that should be provided by all instances of Arcademis. The three main actors of service-oriented architectures are depicted in Figure 4 (b).

Arcademis provides an interface to client applications having access to the discovery agency; another interface is used by service providers. 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. Service providers register themselves using a `publish` operation, while clients look for distributed objects by means of a `find` operation.

3.8. Remote Object Activation

The activation of remote objects in Arcademis is implemented by a component called `Activator`. This component allocates the resources the server needs in order to process remote invocations. For example, it initializes data structures internal to the middleware and creates threads to wait for remote calls. 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

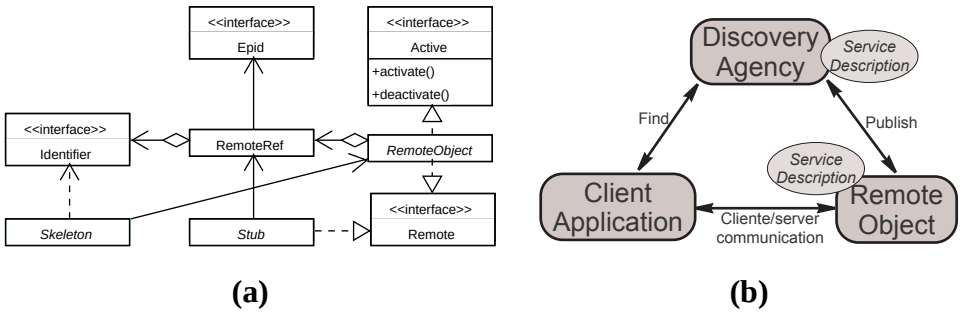


Figure 4: (a) Remote object representation. (b) Service-oriented architecture.

be explicitly invoked by application developers or it may be automatically called during instantiation of remote objects.

3.9. Invocation Policy

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 remote calls' arguments; (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 functionalities to a invoker, Arcademis provides an *invoker decorator*, which is used in the same way as the channel decorator described in Section 3.1. Examples of capabilities that can be aggregated to invokers are: caches (to avoid the transmission of calls already requested), 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 the client does not stay blocked during the remote processing. In this case, results of remote invocations are inserted into a buffer that the client can inspect afterwards. Because invoker decorators only affects the client side of a distributed application, the chain of invokers can be modified during execution time.

3.10. Dispatching Policy

In Arcademis, the overall structure of servers is defined by a component called `Dispatcher`. The implementation of this component determines, for example, if calls are passed directly to the skeleton or to other components. An example of server structure is presented in Figure 5. In this example, there are three active objects: the activator, the scheduler and the response sender. Call descriptors are inserted into a queue and ordered by the scheduler, before being passed 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.

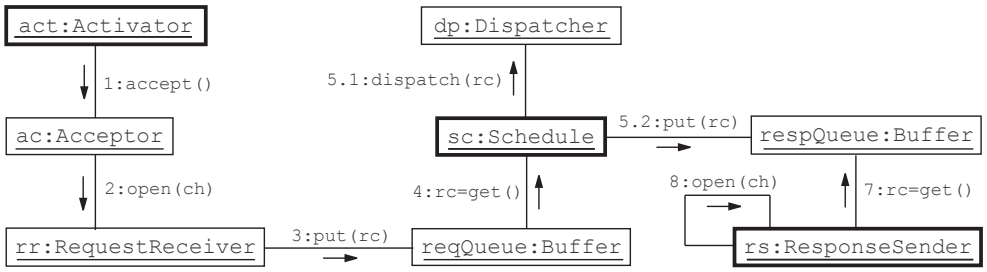


Figure 5: Representation of the main components of Arcademis.

3.11. Priority Policy

Arcademis supports the establishment 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, to clients and to servers' end points. In the last case, it is assumed that servers may receive request in more than one endpoint. Besides changing the scheduler, the implementation of some priority policies also requires changes in other components. For example, in order to assign each method a different priority, it is necessary to modify the implementation of stubs.

4. RME: RMI for J2ME

In order to validate Arcademis, this framework has been used to derive a remote invocation service for Java 2 Micro Edition, a Java distribution that targets resource constrained devices such as cell phones and palmtops [Riggs et al., 2001]. The J2ME platform is divided into different configurations, each of them proper to a specific family of devices. A J2ME configuration defines a Java Virtual Machine, a set of libraries and the Java capacities that are available to devices that meet the minimum set of requirements stipulated by that configuration. Presently, J2ME provides two main configurations: CDC (*Connected Device Configuration*) and CLDC (*Connected, Limited Device Configuration*). CDC groups devices that can afford at least 2MB of memory and persistent network connections, often based on TCP/IP. This configuration provides the application developer with almost all the features found in the standard Java development kit, such as reflection and a complete set of I/O libraries. The CLDC configuration is suitable to more limited devices with memory budgets of no more than 500 Kilobytes, low bandwidth and intermittent network connections. The CLDC configuration does not feature, for instance, the primitive types *float* and *double*, neither computational reflection. Therefore, because the Java RMI serialization mechanism is based on reflection functionalities, this platform cannot be employed in the CLDC configuration.

The proposed service, called RME (*RMI for J2ME*) [Pereira, 2003], provides J2ME's CLDC configuration with a remote invocation service. The main elements involved in the execution of a remote call are depicted in Figure 6. RME is a synchronous

service, meaning that the client application remains blocked while a remote operation is being processed. In the server side, the activator and the request receivers are active objects, being a new thread created for each incoming connection. This arrangement permits to separate the thread in which connections are received (the acceptor's thread), from the threads in which connections are handled (the request receivers' threads). In the presented scheme, `AppStub` and `AppSkeleton` are automatically generated instances of the stub and the skeleton, respectively. In order to allow this automatic generation of components, RME provides `rmec`, a tool that produces source code from the implementation of remote objects. `Rmec` makes use of reflection, instead of traditional parsing, to generate code. It is possible to customize `rmec` to assign a different invoker to each generated method, in order to associate different invocation tactics with them. In Figure 6, for example, `rmec` has assigned the method `m()` an instance of `TwoWayInvoker`.

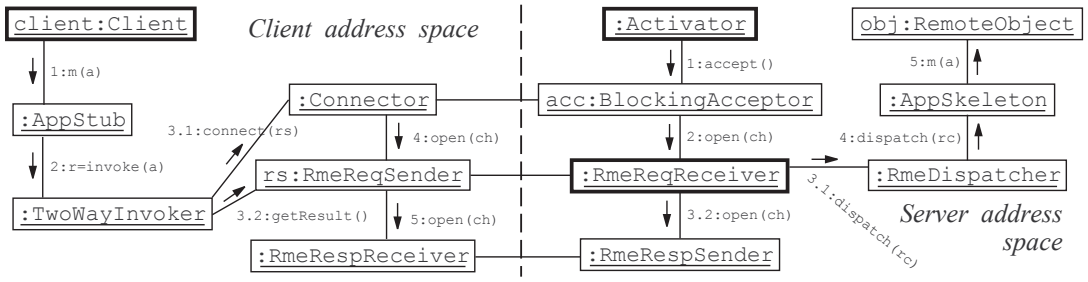


Figure 6: Architecture of RME.

The implementation of `TwoWayInvoker` reuses connections across successive calls and provides to the application developer several different tactics for remote invocation: it is possible to use a cache for storing the result of idempotent calls, it is possible to group several calls together in a single invocation, in order to take better benefit from the available bandwidth and it is possible to create separate threads to carry on remote calls. Two different semantics of call processing have been implemented for RME: best-effort and at-most-once. The adoption of each of them is just a matter of assigning to the ORB the proper service handler factory. Performance tests show that providing an at-most-once guarantee level to the application adds no more than .5 percent of time overhead when compared to the best-effort semantics, although the first strategy requires substantial space for storing identifiers in the server side [Pereira, 2003]. RME uses the TCP/IP transport protocol for data transmission. The communication protocol adopted by RME is named RMEP (*RME Protocol*), and it defines seven different types of messages: *call*, *return*, *ping*, *ack*, *inq*, *load* and *mult*. The *Call* message contains the description of one remote method invocation, what includes 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. Finally, messages of the *mult* type contain several remote calls grouped in a single package.

RME gives to the application developer a programming syntax similar to that 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 have to implement that interface and have to extend the `RmeRemoteObject` class. Figures 7 (a) and (b) show

an example of remote interface and its implementation. Although distributed objects' methods may be invoked remotely, their implementations do not present any particularity for accessing the subjacent network. The middleware transparently gives to the application developer the means of calling those methods across the network. In the given example, the remote method simply sums two integer numbers and returns the operation's result. The server code responsible for the remote object initialization is shown in Figure 7 (c) and the client that invokes a remote method can be seen in Figure 7 (d). Any distributed application based on RME has to determine an ORB customization before starting its execution, what is done by an instance of the `RmeConfigurator` class. The `configure` operation determines the set of component factories that will be associated with the ORB. The discovery agency of RME is implemented by the `RmeNaming` class, and it does not use the interface provided by Arcademis. Instead, it defines the same set of methods provided by the class `java.rmi.Naming`, the lookup service implementation of Java RMI. Because RME targets resource constrained devices, stubs are created by `RmeNaming` according to the *flyweight* design pattern [Gamma et al., 1994]: before creating a stub, the discovery agency checks if there is already an instance of stub that points to the same remote object. If there exist such instance, a reference to it is returned, instead of a reference to a new stub.

```
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 7: (a)Remote Interface. (b)Remote Object. (c)Server. (d)Client.

Some tests have been executed in order to evaluate the performance of the RME implementation. The execution environment consists of a J2ME emulator whose virtual machine (KVM) can execute 100 bytecodes per millisecond. The server and the client emulator were executed in two Pentium 4, with 2.0GHz of clock and 512MB of available memory. The computers were connected by a 10Mb/s Ethernet LAN. The remote methods used in the test are shown in Figure 8 (a). All those methods throws `ArcademisException`, but the declarations have been omitted due to space constraints. In order to determine an upper limit of efficiency, it was implemented a socket-

based application in which clients and servers simply exchange packages of the same size of those used by the RME methods. The average number of requests accomplished per second is presented in Table 8 (b). Each of these values has been obtained as the average of 10 series of 50 remote calls. Because the emulator executes to few instructions per time unit, the serialization of structured types takes considerable time; hence, the methods that pass and return more complex objects are slower than the corresponding upper bound. A comparation between the Java RMI and an implementation of RME for the J2SE environment can be found at [Pereira, 2003]. When processing simple calls, Java RMI is more efficient than RME; however, the presented system surpasses the Sun's implementation when necessary to handle methods that uses structured types, because, while in Java RMI the serialization algorithm is based on computational reflection, in Arcademis it is directly implemented by the application developer; therefore, can be performed faster.

```
import arcademis.*;

public interface MethodSet
extends Remote {
  public short getShort();
  public char getChar();
  public int getInt();
  public long getLong();
  public String getString();
  public String[] getStrs();
  public String passBytes(byte[] b);
  public String passShorts(short[] s);
  public String passChars(char[] c);
  public String passInts(int[] i);
  public String passLongs(long[] l);
  public String passStrs(String[] s);
}
```

(a)

método	RME	upper bound	RME/u. bound
getShort	5.08	5.11	0.99
getChar	5.05	5.12	0.98
getInt	5.06	5.11	0.99
getLong	5.02	5.07	0.99
getString	4.75	5.01	0.95
getStrs	2.96	5.00	0.59
passArgs	2.57	5.02	0.52
passBytes	4.48	5.05	0.90
passShorts	4.26	5.02	0.85
passChars	4.11	5.04	0.82
passInts	3.93	5.08	0.78
passLongs	3.03	5.04	0.61
passStrs	2.12	4.97	0.43

(b)

Figure 8: (a) Interface for performance tests. (b) Performance results: requests/s.

5. Conclusion

This paper has presented Arcademis, a framework for middleware development and one instantiation of it named RME, a middleware system that provides a remote invocation service to the CLDC/J2ME plataform. This research brings forward contributions in methodological and practical fields. First considering the methodological contributions, the paper has presented an analysis of the main constituents of object-oriented middleware architectures, which have been grouped in eleven independent parts. In addition to this, it has defined different ways in which these components can be customized and how such configurations can be accomplished in Arcademis. In practical terms, this research yielded a set of Java classes and interfaces that implement several functionalities required in an object-oriented middleware platform and describe the overall structure of such systems. Another practical result is the derivation of RME.

Arcademis is a general and flexible framework. General because it allows the development of object-oriented middleware for the three main platforms of the Java language: J2ME, J2SE and J2EE. In order to evince this fact, two versions of RME have been implemented: one for J2ME, presented in this paper, and another targeting J2SE [Pereira, 2003].

Arcademis is flexible because every instance of it is ultimately defined by a set of independent object factories associated with the ORB. It is possible to alter a whole aspect of the middleware by just changing the factory that creates the components responsible for that behavior. For instance, RME provides several options that can be configured this way, such as the call semantics, the transport protocol and the invocation strategy. The factory-based design has also the advantage of allowing the middleware to use just the components it will effectively need. This design allows the use of Arcademis in scenarios where more monolithic platforms would not be operational. RME, for instance, is used in an environment where the traditional implementation of Java RMI can not be employed.

Different research threads may be originated from Arcademis. One possible direction of future work is to derive from the framework middleware platforms that do not follow the object-oriented model, such as tuple space-based or message-oriented systems. Finally, the code of Arcademis and RME can be freely downloaded in the URL: `http://www.dcc.ufmg.br/llp/arcademis`.

References

Brose, G. (1997). JacORB: Implementation and design of a java ORB. In *IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*.

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

Johnson, R. E. (1997). Components, frameworks, patterns. In *SIGSOFT Symposium on Software Reusability*. ACM.

OMG (1999). CORBA IIOP 2.3.1 Specification. Technical Report 99-10-07, OMG.

Pereira, F. M. Q. (2003). Arcademis: Um arcabouço para construção de sistemas de objetos distribuídos em Java. Master's thesis, Universidade Federal de Minas Gerais.

Riggs, R., Taivalsaari, A., and VandenBrink, M. (2001). *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. Addison Wesley.

Román, M., Kon, F., and Campbell, R. (2001). Reflective Middleware: From Your Desk to Your Hand. *Distributed Systems Online*, 2(5).

Schmidt, D. (1997). *Acceptor-Connector: Design Patterns for Initializing Communication Services*, chapter 12, pages 191 – 206. Addison-Wesley.

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).

Singhai, A. (1999). *Quarterware: A Middleware Toolkit of Software RISC Components*. PhD thesis, University of Illinois.

Sun (2003). Java RMI home page. http://java.sun.com/j2se/1.4.1/docs/guide/rmi/spec/rmi-TOC.html – last visit: February 2004.