Universidade Federal de Minas Gerais
Instituto de Cincias Exatas
Departamento de Cincia da Computao
Laboratório de Linguagens de Programao

A Three Layer View of Arcademis

Fernando Magno Quinto Pereira
Marco Tlio de Oliveira Valente
Roberto da Silva Bigonha

– Technical Report –
LLP 003/2003

e-mail: {fernandm,mariza,bigonha}@dcc.ufmg.br,mtov@pucminas.br

– Dezembro de 2003 –

Abstract

This technical report presents Arcademis, a framework for object-oriented middleware development. This framework, which has been implemented in the Java programming language, consists of a set of abstract classes and interfaces that define the general architecture of middleware platforms. The main objective of Arcademis is to allow the implementation of non-monolithic and easily reconfigurable middleware systems that can be customized to fit different scenarios. In order to illustrate the use of the framework, the paper also presents RME, a remote method invocation service that has been derived from Arcademis to support distributed applications in the CLDC configuration of Java 2 Micro Edition.

0.1 Introduction

Distributed applications are each time more necessary and their demand has increased continuously since their appearance. In order to evince this fact it suffices to point that the world-wide Internet community increases by 17% each year, having presently more than six hundred million users [6] who communicate and share resources by means of distributed systems. In addition to this, examples of distributed applications can also be found in more restrict networks which belong to institutions such as universities or private companies. Such systems are developed with several different purposes, for example: making information available, providing communication between users, using idle computational resources productively and replicating data to improve its accessibility and security [3]. Although these applications can be quite different in terms of aims and implementations, all of them have in common the capacity of integrating distinct computers.

The design and implementation of distributed applications is generally more difficult than the development of local systems because in the former case the programmer has to deal with distinct machines, and, consequently, with the heterogeneity of computer architectures and communication protocols. In order to provide the application developer a set of abstractions that facilitate the design and implementation of distributed programs, it has been introduced the concept of middleware system: a software layer that is interposed between the operating system and the applications [5]. The middleware's main objective is to prevent the programmer from having to deal with the low-level primitives that the operating system provides for data transmission. For example, middleware systems can make the programmer oblivious to different techniques of number representation such as little endian or big endian. If it were not by these platforms, there would be cases when the programmer would have to explicitly convert numbers from a format to another when passing them between different computers.

There are several kinds of middleware platforms [9], such as message passing systems or tuple-space based systems; however, object-oriented middleware seems to be the most popular platforms at the present time. Examples of such platforms are CORBA [11], Java RMI [17] and .NET [10]. These middleware have in common the fact of being based on the model first introduced by the *Network Objects* of Modula-3 [1]. According to that model, the programmer can invoke methods on remote objects as though they were present in the local address space. This is a really powerful abstraction because the application developer can take all the benefits from the object-oriented paradigm while programming in a distributed environment.

The most popular middleware architectures are complex systems originally devised to fit the necessities of stationary computer networks plenty of resources such as memory and bandwidth. These platforms are monolithic software that barely support customization and adaptation [5]. The advent of mobile computing brought on the necessity of developing applications targeting resource constrained devices, such as cell phones and palmtops. A traditional mono-

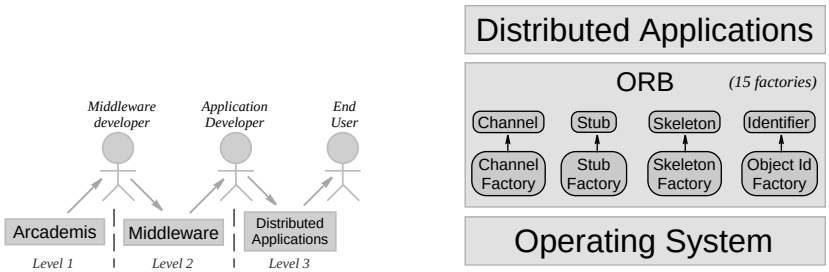
lithic architecture cannot be properly employed in this environment because, in addition of being too heavy for the requirements of mobile devices, its unused functionality cannot be removed from the whole platform in order to make it lighter. For instance, a CORBA implementation such as VisiBroker or Orbix that occupy tens of megabytes of memory cannot be deployed in a cell phone with a total memory footprint of 128-KBytes.

In order to support the implementation of modular and highly customizable middleware platforms, this article introduces Arcademis: a Java based framework for middleware development. A framework, essentially, is a reuse technique, which is normally represented by a set of abstract classes and interfaces that, together, describe the skeleton of an application that can be customized by the programmer [8]. The main purpose of Arcademis is to be as flexible as possible, allowing the deployment of middleware platforms that fits several different sets of requirements. Therefore, instances of Arcademis can be customized to provide distributed applications with just the very functionality that they will need. For instance, the specification of a particular application can state that cell phones will not be used as servers. In this case, Arcademis can be used to derive a middleware system that only provide client capabilities to developers.

This paper also presents RME [12], an instance of Arcademis that has been developed to provide a particular configuration of J2ME [14], called CLDC, with a remote invocation method service, similar to Java RMI. J2ME is a Java distribution that targets resource constrained devices, and Java RMI is the standard remote invocation service provided by the Java language. Although both Java RMI and RME, offer application developers essentially the same functionality, these systems differ fundamentally with respect to the technique used to accomplish object serialization. Such difference make it possible to use RME in an environment where Java RMI can not be put to work. To the best of the authors knowledge, there is no other remote invocation service that can be used in the CLDC configuration; hence, the existence of RME can be regarded as a concrete contribution to the research presented in this paper.

0.2 Architecture of Arcademis

The implementation of Arcademis is based on a number of design patterns, such as object factory, decorator, singleton and proxy [4]. These patterns are programming techniques that can be employed for solving families of related problems. Therefore, a design pattern can be be described by the specification of a problem, its solution and the context in which that solution works [8]. The utilization of these techniques in the design of Arcademis brings several advantages. First, the patterns make it easier to document and organize the framework's implementation, because they provide a common vocabulary for describing the components and the interactions among them. Secondly, the patterns contribute for enhancing the flexibility of the framework and of the middleware platforms derived from it. This second benefit comes from the



(a) **(b)**

Figure 1: (a) The programming level views. (b) The ORB organization.

fact that several design patterns have being engendered in order to allow the implementation of more modular systems that could be easily modified.

A distributed system built on top of Arcademis has three main constituents or programming levels. The first of these levels is composed by the elements that constitute Arcademis. Essentially these are abstract classes and interfaces, although the framework also provides concrete components that can be used without further extensions. The second level is formed by the concrete middleware platform, obtained as an instance of Arcademis. The framework postpones to this level decisions such as the communication protocol and the serialization strategy that will be adopted, for example. Finally, the third programming level comprise all the components that will provide services and abstractions to users that are not necessarily programmers. These components constitute what is normally called distributed applications, such as e-mail systems or virtual bookstores, for example. A diagram depicting the relation between the programming levels and their users is presented in Figure 1 (a).

Each instance of Arcademis has a central component called ORB. This element is implemented as a *singleton*, a design pattern that guarantees that, given a class, there will exist just one instance of it per application [4]. Besides being implemented as a singleton, the ORB can also be characterized as a set of *object factories*. An object factory is another design pattern [4] that is used to generate instances of objects. The main advantage of this pattern is to make it easier to change a component's implementation without interfering in other parts of the system. For instance, according to the Arcademis's specification, all the communication channels are created by an object factory. When necessary 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 channel's interface keeps the same, the other platform's components need not to be altered. Figure 1 (b) pictures the general organization of any distributed system based on Arcademis.

Arcademis has been originally devised to permit the development of object-oriented middleware platform. According to this model, a client object relies on intermediate components in order to invoke methods on remote objects. These

intermediate elements are called *stubs* and *skeletons*. Although the application developer has the illusion the methods are being locally processed, actually every remote call is transmitted by the stub to the skeleton and then to the implementation of the remote object. The method's result is sent in the opposite way. In order to best describe the Arcademis architecture in view of such model, the framework's structure can be divided into three main layers: the stub/skeleton layer, the remote reference layer and the transport layer. These parts are discussed in the remainder of this section.

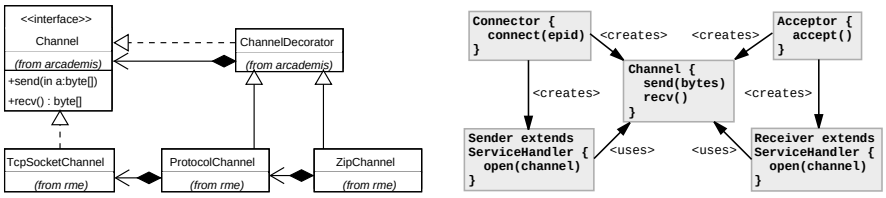
0.2.1 The Transport Layer

The transport layer comprises all the components responsible for transmission of data between hosts and by connection establishment. This is the only layer which is in direct contact with the operating system, since it needs to deal with the raw sequences of bytes that represents information to be transmitted. The main components of the transport layer are the channel implementation, the connection server, the communication protocol and the components responsible for connection establishment. Arcademis allows the middleware developer to configure all of these elements.

The most basic element of the Arcademis's transport layer is called a *channel*. This component is responsible for transmitting byte sequences between clients and servers. The framework does not assumes the utilization of any specific transport protocol, and possible implementations can be based, for example, on UDP, TCP, HTTP or SOAP. In order to add further functionality to a channel, Arcademis uses a design pattern called *decorator* [4], which provides a way to modify the behavior of individual objects without having to create new derived classes. A channel decorator is an object that extends that class and, in addition to this, has an attribute of the channel type. As a subclass of channel, the decorator can overwrite some of that component's methods in order to aggregate further capabilities to them.

Figure 2 (a) shows an example of composition of decorators. `ZipChannel` compress messages in order to take better benefit from the available bandwidth and `ProtocolChannel` implements the middleware communication protocol, that defines the kinds of messages available for performing all the services offered by the platform. The class labeled `TcpSocketChannel` is one of the concrete components provided by Arcademis. The same chain of capabilities could have being built by means of inheritance, but, in that 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 necessity of modifying other's components code. Simple inheritance does not afford such flexibility.

The process of connection establishment, in Arcademis, has been implemented according to the *acceptor-connector* design pattern [15]. 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



(a) **(b)**

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

connector is responsible for contacting the acceptor when necessary to set up a channel between two hosts. Once the process of connection establishment has been completed, the resulting channel is passed to a service handler, which will send and receive messages through it according to the needs of a particular application. One of the advantages of this design pattern is the possibility of configuring different connection strategies without the necessity of modifying the service handlers' code. Possible strategies include synchronous and asynchronous connection establishment and the use of caches for reusing channels.

0.2.2 The Remote Reference Layer

The remote reference layer contains all the components responsible for the location and identification of remote objects distributed across a computer network. Its most important element is called a *remote reference*. This component provides client applications the means to contact the remote object it represents; hence, it contains two basic values: the first of them is the location of the remote object in the distributed environment, and the other is an identifier, which is used to distinguish a remote object from another. Another function of remote references is to define the semantics of common operations pertaining to every object, such as `toString`, `equals` and `hashcode`.

The framework represents the unique identifier and address of an object by the classes `Identifier` and `EndPointIdentifier` respectively. Again, Arcademis does not determine any specific implementation for these components. For instance, remote addresses can be defined as ordered pairs formed by a host name and a port number, or they can encapsulate the location of a shared memory address. 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 identifiers should grant that in the whole distributed network there will be no two distinct remote objects holding equal identifiers.

Another element of the remote reference layer is the `RemoteObject` class, which should be extended by every object whose methods can be remotely invoked. That is an abstract component that has to be implemented in every instance of middleware platform derived from Arcademis. Among the methods

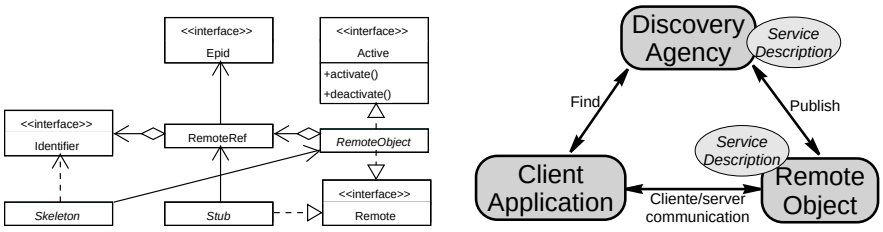


Figure 3: (a) Components of the remote reference layer. (b) Service-oriented architecture.

of `RemoteObject`, two deserve special attention: `activate` and `deactivate`. Both are abstract methods, and the first of them defines how an object will be made ready for receiving and processing remote calls. The second operation permits to release to the operating system the resources employed by the remote object for handling requests. The relations among the main participants of the remote reference layer is shown in Figure 3 (a).

The last component of the remote reference layer is the *discovery agency*. The great majority of middleware platforms obtained from Arcademis can be described as service-oriented architectures [2]. This model involves three different actors: service providers, service requesters and discovery agencies. Service providers are represented by the implementation of remote objects, whereas the requesters are represented by clients in general. The discovery agency, or name service, is a independent element that should be provided by all the Arcademis's instances. The three main constituents of the service-oriented architecture are depicted in Figure 3 (b).

Arcademis does not define a standard interface for name services because it can vary considerably among applications. For instance, the Java RMI platform defines a name server based on five operations: `lookup` and `list` for discovering names and `bind`, `rebind` and `unbind` for registering and unregistering services. According to that model, quests are based on the strings bound to remote objects in the directory of names. On the other hand, it is possible to define a name server that associate to remote references interfaces or service descriptions instead of names. Therefore, the stipulation of a specific type of discovery agency would impact on Arcademis's flexibility. In spite of that, the framework defines a basic name server interface as part of its collection of concrete components. Such functionality provides two operations: `find` and `publish`, but they are declared in different Java interfaces located in different packages. That is because clients applications normally do not use the *publish* operation, so clients should not have to import the package that contains the `find` functionality.

0.2.3 The Stub/Skeleton Layer

In a distributed system, normally clients and servers are not located on the same node; hence, in order to allow the communication between them, it is necessary to provide each host with a local representative of the entity not physically present. The stub is the local representative of the server in the client address space, and the skeleton is the local representative of the client in the server address space. The stub may be described by the *proxy* design pattern [4], which is used in situations when it is necessary to provide a surrogate for an object. As a proxy, the stub defines all the remote methods of the object it represents. When the client application performs a remote method invocation, it is actually calling a method on the stub, that will take the responsibility of delivering the call to the actual implementation of the remote object. The stub counterpart in the server's address space is called skeleton: a component that can be described by the *adapter* design pattern [4]. The skeleton's main function is to receive requests describing method invocations and to assign them to the component responsible for their processing. After the method's processing, the skeleton sends the result of that execution to the client side stub.

The skeleton and the stub communicate by means of four different service handlers that, together, constitutes a design pattern called *request-response*. These service handlers are called *request-sender*, *request-receiver*, *response--sender* and *response-receiver*, and Figure 4 (a) shows the relation between them. The major advantages of this pattern is the possibility of easily configuring the politics of thread utilization in the client and in the server spaces. In addition to this, the request-response pattern facilitates the customization of the remote call semantics.

In the Arcademis framework, clients and servers can be customized according to different strategies of thread utilization. For instance, a client can be defined to be synchronous or asynchronous. In the former case, the client application remains blocked during all the process of remote method execution, until it receives the call's result or a network error takes place. In the asynchronous case, the client does not stay blocked while the call is being processed by the server. This strategy can be used to increase the client's throughput [3]. Asynchronous method invocation also gives room for further optimizations. One such optimization is to bufferize several requests so that all of them can be sent in a single package. The use of the buffer allows a better utilization of the available bandwidth.

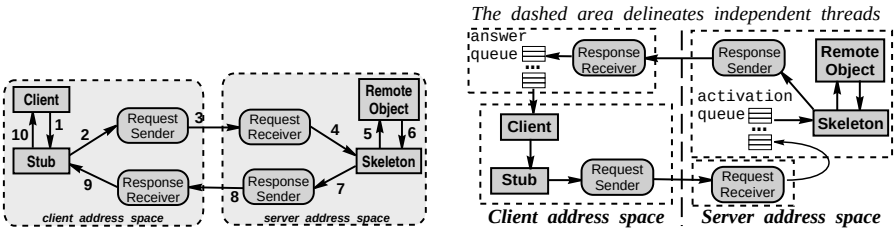
Arcademis also permits the developer to define different strategies of thread utilization for the server side of the middleware. The server can be configured to run in a single control thread or in several threads. In the second case, it is possible to distribute tasks among threads in different ways. For instance, a new, independent thread, can be allotted to each incoming method request, or to each connection created with the server, or to each client that is accessing the server. In addition to this, separate threads can be designated to some of the server's components, such as the acceptor or the response-sender. The request-response pattern also allows the customization of structural aspects of

the server architecture. For example, the request-receiver can directly deliver method requests to the skeleton, or they can be inserted into a queue, where a scheduler running in a separate thread is responsible for determining the priority in which remote requests are processed.

Figure 4 (b) depicts an example of client/server configuration, where method invocations are asynchronously delivered to the server. The server architecture obeys the *active object* [16] design pattern. This pattern suits well the necessities of a remote invocation method service because it decouples an operation execution from its calling. According to this technique, requisitions of method execution are passed to an object that resides in its own thread of control by means of a data structure called *activation queue*. The order in which tasks are retrieved from the queue and sent to the active object is determined by a component known as *scheduler*. Arcademis provides the developer a factory of queues and several different implementations of such structures. Two instances of them are visible in the figure: the `answer queue` and the `activation queue`. A scheduler factory is also provided by the framework together with a default implementation.

The utilization of the active object pattern affords several possibilities of reconfiguration to the server structure. Firstly, it allows the adoption of different strategies for thread's management: the active object may be executed as a single thread or as a pool of threads. Secondly, it provides the middleware developer different ways to determined the priority of method execution. For instance, in order to implement the *shortest-job-first* algorithm, the stub can assign each method a constant priority that the developer determines according to that method's complexity. Simpler methods are given higher priorities because they tend to be processed in a shorter period of time. A second scheduling algorithm, called *first-in, first-out*, can be implemented by ascribing to each method a priority based on the time its requisition has been inserted onto the activation-queue. Requisitions may also be given different priorities based on the requester's identity. In this case each client has a specific priority, that may be defined by several different factors, such as the frequency in which the client issues remote method invocations. In this case, clients that access the server more frequently can be given a higher or lower priority, depending on the adopted algorithm. The use of threads and the distribution of priorities among remote invocations can be customized by changing the implementation of the scheduler component of the framework.

The client shown in Figure 4 (b) performs asynchronous calls, what means that is does not stay blocked while remote operations are being processed by the server. An independent thread is created in the client's space for receiving the results of remote calls. Such results are stored in a queue that can be inspected by the client application at any time during its execution. Because invocations are made asynchronously, messages can be grouped in a buffer, in order to be sent together. The buffer utilization delays the transmission of method requests; hence, it may increase invocation's latency. On the other hand, it reduces the number of packages sent through the transport layer, what can improve application's throughput. The event that causes the delivery of



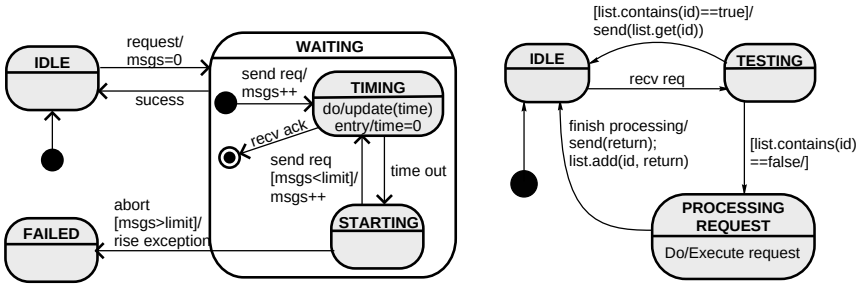
(a) **(b)**

Figure 4: (a) Request-response. (b) Example of server configuration.

requests stored in the buffer can be customized in a number of ways. A simple strategy is to send the stored messages when its quantity reaches a certain limit. The number of bytes in the buffer may also be used as the triggering criterion for sending the grouped messages. In addition to these two strategies, the buffer's content should be sent to the server within regular time intervals, in order to diminish the impact of the buffer over the application's latency.

Another main benefit of the request-response pattern is to allow the parameterization of remote invocation's semantics. The semantic of a remote call defines the level of reliability the middleware platform provides to the client application. The chances of a method invocation not been processed in the expected way in a distributed environment is orders of magnitude greater than in the local case. The three most popular invocation semantics are called *best-effort*, *at-most-once* and *at-least-once* [3]. The first of them do not offers the client any guarantee regarding the remote call's processing. It may be executed one time, several times, or no time at all. The semantics know as at-most-once gives the client application the guarantee that the remote call will be processed no more than one time, but it may be not executed. Finally, the at-least-once semantics affords the client application the guarantee that remote calls will be necessarily executed one or more times.

The configuration of most of the aspects of middleware's internal structure discussed in this section are restricted to the implementation of the four service handlers that constitute the request-response design pattern. For instance, in order to group messages in a buffer for improving application's throughput, it suffices to alter the code of the request-sender. Different implementations of the request-receiver yields different politics of thread utilization, such as thread-per-request or thread-per-connection. In addition to this, the three distinct semantics of method invocation can be implemented by changing the structure of just the request-sender and the request-receiver. Figure 5, for example, shows the state machines that characterize the implementation of the at-most-once semantics. According to that scheme, the request sender, after sending a method invocation request, starts a time counter, and, if a certain interval has elapsed before it receives any response for the remote call, it will repeat this process. After a given number of calls without answer, the client application assumes



(a) **(b)**

Figure 5: At-most-once call semantics: (a) request-sender state machine. (b) request-receiver state machine.

that the server is not achievable, and an exception is throw, denoting this fact. The request receiver, on the other hand, keeps the calls' identifiers in a list, in order to discard repeated requests. In the list, identifiers of method invocations are associated to their return values; therefore, if a repeated request is received, the server can answer it without further processing.

Serialization of Objects in Arcademis

Object serialization is the process of converting the object's internal state into a raw sequence of bytes in such a way that the original object can be recovered from it. This mechanism allows an object to be recorded in a file-system or transmitted along a network. In the Java RMI platform, objects are passed as arguments or return values of remote invocations as serialized data. Because an object usually contains references to others, in order to serialize it, it may be necessary to traverse the graph of references that constitutes its internal state, so that no information be lost in the process. In the Java programming language, such task is accomplished by means of a mechanism called reflexivity. Computational reflection provides a program the knowledge about parts of its internal structure, such as the type of a variable or the list of methods pertaining to a certain class. The use of reflection makes it possible to automate the serialization process in such a way that the programmer is unaware of it; hence, it saves a lot of effort that, otherwise, would be spent in the manual conversion of objects and primitive data into byte sequences.

Although the serialization strategy adopted in the Java language offers the programmer a great level of abstraction, it has some limitations. The most important of them is the existence of environments where reflexivity is not available. For example, the CLDC configuration of the J2ME platform does not provide all the reflexivity capacities present in other editions of the Java language. As a consequence of this limitation, it is not possible to employ Java RMI, as it is currently implemented, in that scenario. In addition to this, RMI's architecture is not enough modular to allow the adoption of another serialization

```
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();
  }
}
```

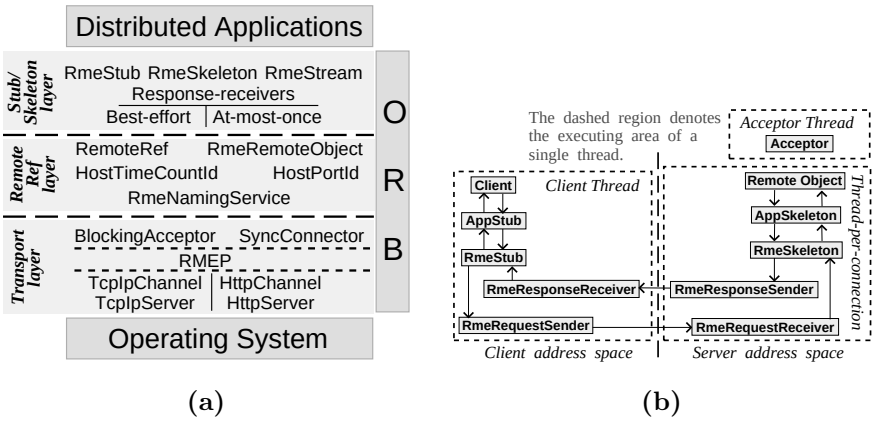
Figure 6: Example of serializable class.

strategy that does not require the availability of computational reflexivity.

It is desirable that middleware platforms obtained from Arcademis be used in all the three main distributions of the Java language: J2ME, J2SE and J2EE. In order to accomplish this requirement, Arcademis does not relies on particular characteristics of any of the platforms, what includes the reflection mechanism. The strategy of object serialization adopted by Arcademis consists in transferring to the application developer the task of implementing the serialization routines of all the classes whose instances may need to be serialized. With this objective, Arcademis defines the interfaces `Marshalable` and `Stream`. According to the Arcademis specification, serializable objects are instances of `Marshalable`. This interface declares two methods: `marshal` and `unmarshal`. The first of them describes how an object is converted into a sequence of bytes, whereas the second determines how the state of the object can be recovered from that sequence. The `Stream` interface specifies the serialization protocol, by means of a collection of methods for reading and writing byte sequences. Although this interface can be implemented in different ways, in essence, any implementation encapsulates an array of bytes. An example of class that implements `Marshalable` is presented in Figure 6.

0.3 RME: RMI for J2ME

In order to validate the Arcademis framework, it has been used to derive a remote invocation service for Java 2 Micro Edition [14], a Java distribution that targets resource constrained devices such as cell phones and palmtops. The J2ME platform can be divided in 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 for a group of devices that meet the minimum set of requisites stipulated by that configuration. Presently, J2ME provides two main configurations: CDC and CLDC. The first configuration, named CDC (*Connected Device Configuration*), groups devices that can afford at least 2MB of memory and persistent network connections, often using TCP/IP. This configuration provides the application developer with



(a) **(b)**

Figure 7: (a) Overview of the RME architecture. (b) Use of threads in RME.

almost all the features found in the standard Java development kit, such as reflexivity and a complete set of I/O libraries. The other configuration, called CLDC (*Connected, Limited Device Configuration*) targets more limited devices, generally mobile and battery-operated, with memory budgets of no more than 500 Kilobytes, low bandwidth and intermittent network connections. The CLDC libraries contains classes that are not present in the J2SE libraries, but, in general, this configuration provides to the application developer far fewer capabilities than the standard development kit. It does not afford, for instance, the primitive types *float* and *double*, neither computational reflexivity, although such capacities are present in CDC. As told in Section 0.2.3, Java RMI can not be used in CLDC due to this configuration not offering the reflexivity functionality of the Java language, that is necessary for object serialization.

The proposed service, called RME (*RMI for J2ME*) [12], targets the CLDC configuration, and intends to permit the development of distributed applications that communicate by means of a remote method invocation mechanism. An overview of the RME architecture is presented in Figure 7 (a). The use of threads in the client and server spaces as well as the main elements involved in the execution of a remote method invocation are depicted in Figure 7 (b). RME is a synchronous service, meaning that the client application remains blocked during all the time in which a remote call is being processed. The server creates a new thread to handle every new incoming connection, and the acceptor is executed in a separate thread. In that scheme, `RmeStub` and `RmeSkeleton` are subclasses of `arcademis.Stub` and `arcademis.Skeleton` respectively, and `AppStub` and `AppSkeleton` are automatically generated instances of these local representatives. In order to allow automatic generation of stubs and skeletons RME provides `rmec`, a tool that produces the source code of these components from the implementation of a remote object.

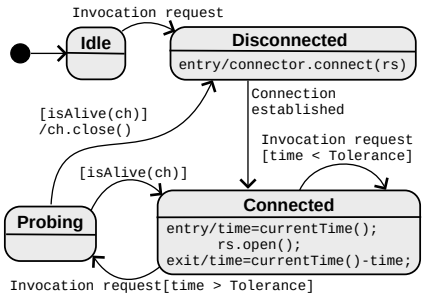
Figure 8 (a) shows the statechart diagram that characterizes the RME stub. This diagram evinces the fact that the client tries to take benefit of the same

connection successive times, in order to improve its efficiency. The stub establishes a connection with the server when it first receives an order to perform a remote method invocation. The next call can be handled in two different ways. If there is not elapsed much time since the last remote invocation, the previously used channel is utilized again. In this case, the implementation of RME guarantees that there will be a request-receiver listening that channel in the server side. On the other hand, after some period of time, if the channel has not been used, RME tries to recycle the connection, in order to save resources and for security reasons. If that is the case, the stub probes the channel with a *ping* message in order to verify whether the channel is still alive, and if there is no answer, a new connection is created with the server, otherwise the old one is reused.

Two different semantics for 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 [13]. RME also provides to the application developer two implementations of the communication channel, each of them using a distinct transport protocol: TCP/IP and HTTP. The second channel is necessary because the first release of CLDC (CLDC 1.3) just supports HTTP connections. The TCP/IP protocol is supported by CLDC 2.0. Again, customizing the channel that will be utilized in RME is just a question of associating the correct factory to the ORB.

The communication protocol adopted by RME is named RMEP (*RME Protocol*), and defines four different types of messages: *call*, *return*, *ping* and *ack*. *Call* messages are used to characterize packages containing the parameters of a remote method invocation. Return values are sent by means of *return* messages. The other two messages are mostly used to probe channels and in order to verify if servers or clients are alive. Every *ping* message forces the receiving entity to answer it with an *ack* package. The protocol specification can be seen in Figure 8 (b). In that grammar, the symbol `Epid` stands for the location of a remote object, and is used as a return address inserted into messages that originates an answer from the receiving entity. Actually the return address is not necessary in the current implementation of RME because the same connection is used for sending and receiving messages during remote method invocation or during a channel test. In an asynchronous implementation, however, the server could use the return address as the destiny location to the values obtained as consequence of remote call processing. Finally, the symbol `Stream` denotes a sequence of bytes originated from the serialization of a list of objects and primitive data.

RME provides the application developer a programming syntax similar to that provided by Java RMI. Remote methods must be declared in a interface that extends the `arcademis.Remote` interface and must declare the possibility of throwing `arcademis.ArcademisException`. The implementation of that interface is a class that should extends `RmeRemoteObject`. Although instances



(a)

$$
\begin{array}{rcl}
message & \rightarrow & header\ version\ msg_{type} \\
header & \rightarrow & \texttt{0x52 0x4d 0x45 0x50} \\
version & \rightarrow & \texttt{0x01} \\
msg_{type} & \rightarrow & msg_{req} \mid msg_{ack} \\
msg_{req} & \rightarrow & \texttt{epid}\ req_{type} \\
req_{type} & \rightarrow & \texttt{call}\ call_{id}\ obj_{id}\ op\ args \\
 & \mid & \texttt{ping} \\
call_{id} & \rightarrow & \texttt{id} \\
obj_{id} & \rightarrow & \texttt{id} \\
op & \rightarrow & \texttt{int} \\
args & \rightarrow & \texttt{Stream} \\
msg_{ack} & \rightarrow & \texttt{return}\ call_{id}\ ret_{val} \\
 & \mid & \texttt{ack} \\
ret_{val} & \rightarrow & \texttt{Stream}
\end{array}
$$

(b)

Figure 8: (a) Statechart diagram of `RmeStub`. (b) RMEP Specification.

of such class will be invoked remotely, its implementation do not present any particularity for accessing the subjacent network other than the fact it extends `RmeRemoteObject`. Figure 9 presents an example of distributed application based on RME. The remote method simply sums two integer number and returns the operation's result. The server code responsible for the remote object initialization is shown in Figure 9 (c) and the client that carries on a remote method invocation can be seen in Figure 9 (d). Both applications contains commands to configure the middleware, that is, to define the factories that will be associated to the ORB. Any distributed application based on RME should determine an ORB customization before starting its execution, what can be done by an instance of the `RmeConfigurator` class.

Some test have been executed in order to evaluate the performance of the RME implementation. The execution environment utilized 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 10 (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 whose client and server simply exchange packages of the same size of that used by the RME methods. The average number of requisitions accomplished per second is presented in Table 10 (b). Each of these values has been obtained as the average of 10 series of 50 remote calls. Because the emulator carries on 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.

Besides the conventional capabilities provided by RME, this middleware contains some additional functionality that suits the necessities of applications executed in the mobile environment. The first of these extra services is a cache that can be appended to stubs, in order to save bandwidth. `Rmec`, the stub generator,

```
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.

```
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)

mtodo	**RME**	**socket**	**RME/socket**
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 10: (a) Performance results: requisitions/s. (b) General vision of RME.

can be customized to associate some methods with the cache, so that, every call on these operations causes their arguments and return value being recorded. If a subsequent invocation presents a list of argument already stored in the cache, than the value associated to that list is returned by the stub, without the necessity of performing a new access to the network. The other additional function that can be used in RME is a *flyweight* factory of stubs. The flyweight design pattern [4] provides an approach to share the same instances between different object references. The flyweight factory keeps a list of stub's instances. When it is given a remote reference to generate a new stub, it verifies whether there is an instance of stub in the list holding that reference, and if that is the case, the old representative is returned instead of a new one.

0.4 Conclusion

This paper has presented Arcademis, a framework for middleware development and one instance of it named RME, a middleware system that provides to the CLDC configuration of Java 2 Micro Edition a remote invocation service. This research brings forward actual contributions in both the theoretical and practical fields. First considering the theoretical contributions, the paper presents an analysis of the main constituents of object-oriented middleware platforms, while grouping them in three layers: the transport layer, the remote reference layer and the stub/skeleton layer. In addition to this, it defines different ways in which these components can be customized and how such configurations can be accomplished. 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, more important, describe the overall structure of such systems. Another practical result from this research is the derivation of RME. This middleware provides the application developer a high level of abstraction because it allows him to take benefit of the power of the object-oriented programming style while making him unaware of objects' real location. Furthermore, to the best of the authors knowledge, there is no other complete implementation of a remote invocation service for CLDC/J2ME, although there is a RMI optional package for CDC/J2ME [7].

This paper concludes with a brief evaluation of the Arcademis framework and RME, taking into consideration three main criteria: generality, flexibility and usability. Starting by the generality, it is important to say that Arcademis is essentially a framework targeting object-oriented middleware development; however, it can be used to derive middleware platforms pertaining to other paradigms. For instance, a complete discussion of how Arcademis can be used in the implementation of *PeerSpaces*, a tuple Space-based middleware is presented in [13, pages 105–106]. Another element that adds favorably to Arcademis's generality is the possibility of using the framework to implement middleware platforms for the three main distributions of the Java language: J2EE, J2SE and J2ME. That is possible because the Arcademis's core components are implemented with classes, interfaces and primitive types that are common to all

the Java editions. Two versions of RME have been implemented: one for J2ME, presented in this paper, and another targeting J2SE, the Java standard development kit. Although this implementation is slower than Java RMI when processing calls that only present primitive types, it surpasses its counterpart when considering calls that use more complex data types due to a faster serialization strategy [13].

The flexibility can be pointed as the main benefit of Arcademis. Every instance of this framework is ultimately defined by a set of independent object factories. It is possible to alter a whole aspect of the middleware by just changing the factory that creates the components responsible by that behavior. For instance, there are two different implementations of call semantics and transport protocol for RME. In order to alter any of these middleware's properties, it suffices to replace the channel factory or the service handler factory in the `RmeConfigurator` class. The factory-based design has also the advantage of allowing the middleware to use just the components it will effectively need. As an example, RME does not employs the queue factory or any of the concrete queue implementations provided by Arcademis. Such components are not even part of the RME library. By the other hand, a platform similar to that presented in Figure 4 would take benefit from these structures. This design allows the use of Arcademis in scenarios where more monolithic platforms could not be put to work. RME, for instance, is used in a environment where the traditional implementation of Java RMI can not be employed. In addition to this, the possibility of giving the middleware just the capabilities it will need allows the generation of platforms having small libraries, what suits the tight requirements of embedded and mobile devices. The client version of RME's libraries has 34.5KB, and an version of this platform, without object factories, has 29.2KB.

Finally, it is important to say that the Arcademis's architecture, based on design patterns, favors its usability, because most of these patterns are already well know by the programming community. In addition to this, the modular structure of the framework allows the modification of some of its parts without the necessity of extending such changes to other parts, what makes it possible to reuse complete middleware platforms in order to address a new set of requirements. In general alterations in components of one specific layer (transport, remote reference or stub/skeleton) do not have any impact over the constituents of the remaining layers. There are, however, exceptions to this rule. For instance, if the addressing strategy, implemented by the `Epid` component of the remote reference layer, is changed, than it might be necessary to alter the channel implementation, a transport layer's element, because connections could have to be bound to another kind of remote address specification.

Different research threads can 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. A second research topic is the parameterization of different distributed garbage collection algorithms in Arcademis. Since the introduction of the garbage collection technique of Modula-3 [1], several distinct strategies have been devised for giving back to the operating system unused

memory and it would be interesting to provide an easy way to customize the algorithm adopted in the middleware platform. Regarding RME, the natural sequence of the present research is to provide this system with asynchronous method invocation. Asynchronous calls suit well the necessities of a mobile system because they allow clients to perform activities while disconnected. In addition to this, asynchronous calls may be used for enhancing the efficiency of applications that demand a large number of message exchanges [3].

Bibliography

[1] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *14th Symposium on Operating Systems Principles (SOSP)*, pages 217–230. Software–Practice and Experience, 1993.

[2] Michael Champion, Chris Ferris, Eric Newcomer, and David Orchard. *Web Services Architecture*. W3C, 2002.

[3] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design*, volume 1. Addison-Wesley, 2nd edition, 1996.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994.

[5] Kurt Geihs. Middleware Challenges Ahead. *IEEE Computer*, 34(6):24 – 31, 2001.

[6] Sabine Graumann. Distribution of Internet users. *Monitoring Information Economics – 6th Factual Report*, 1(6), 2003.

[7] Sun Microsystens Inc. Rmi optional package specification version 1.0, 2003. http://java.sun.com/products/rmiop/ – ltima visita: junho de 2003.

[8] Ralph E. Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[9] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Middleware for Mobile Computing (A Survey). *LNCS*, 2497:20 – 58, 2002.

[10] Piet Obermeyer and Jonathan Hawkins. Microsoft .net remoting: A technical overview. Technical Report 013, Microsoft Corporation, July 2001.

[11] OMG. CORBA IIOP 2.3.1 Specification. Technical Report 99-10-07, OMG, 1999.

[12] Fernando Magno Quintao Pereira, Marco Túlio de Oliveira Valente, Roberto S. Bigonha, and Mariza A. S. Bigonha. Chamada remota de mtodos na plataforma j2me/cldc. In *V Workshop de Comunicação sem Fio e Computação Móvel*. SBC, 2003.

[13] Fernando Magno Quinto Pereira. Arcademis: Um arcabouço para construção de sistemas de objetos distribuídos em java. Master's thesis, Universidade Federal de Minas Gerais, 2003.

[14] Roger Riggs, Antero Taivalsaari, and Mark VandenBrink. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. Addison Wesley, 1th edition, July 2001.

[15] Douglas Schmidt. Acceptor-connector – an object creational pattern for connecting and initializing communication services. In *European Pattern Language of Programs conference*, July 1996.

[16] Douglas Schmidt and Greg Lavender. Active object – an object behavioral pattern for concurrent programming. In *Second Pattern Languages of Programs conference*, September 1995.

[17] 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.