PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 44, NO. 1, PP. 29–49 (2000)

TIME INDEPENDENT INVOCATION IN JAVA CMS[1]

Markus HOF* and Attila ULBERT**

*Computer Science Department
Johannes Kepler University
A–4040, Freistädterstraße 315
Linz, Austria
e-mail: markus.hof@eracom-tech.com
** Department of General Computer Science
Eötvös Loránd University
H–1117 Pázmány Péter sétány 1/D
Budapest, Hungary
e-mail: mormota@elte.hu

Received: 10 Nov. 2000

Abstract

Most object-oriented languages for distributed programming offer a limited number of invocation semantics. At best, they support a default mode of synchronous remote invocation, plus some keywords to express asynchronous messaging. The very few approaches that offer rich libraries of invocation abstractions usually introduce significant overhead and do not support the composition of those abstractions, however, one can never predict the need of the developer, especially in the fast developing and changing mobile environments.

This paper describes a pragmatic approach for abstracting remote invocations in mobile environments and presents the *time-independent invocation* and its formal description. Invocation semantics, such as synchronous, transactional, or replicated, are all considered first class citizens. We completely separate the class definition from the invocation semantics of its methods and we go a step further towards full polymorphism: the invocation of the same method can have different semantics on two objects of the same class. The very same invocation on a given object may even vary according to the client performing the invocation.

Keywords: distributed objects, middleware, abstractions, remote invocation, formal description techniques.

Introduction

The current state of the art for communication paradigms in mobile computing is not really fixed. There is still much discussion about the right way in which a mobile client should interact with the rest of the world. Should we use events or an RPC like mechanism? Should we implement total transparency or should or even do we have to give the users some degrees of freedom? Seen on a more basic level: Should we use synchronous or asynchronous communication mechanisms?

[1]This research work was supported by Grant Nr. FKFP 0206/97

In our opinion, one cannot decide on this issue right now. We think, that even if we could decide, we should not do so, but offer a flexible and fast communication mechanism that provides all the desired degrees of freedom.

Exactly the above mentioned freedom is offered by our composable message semantics framework (CMS) [6] proposed in this paper. The basic communication paradigm offered to the application programmer is the illusion of a dynamically bound method invocation. However, behind the scenes, one has the possibility to arbitrarily change the actual invocation semantic to the desired parameters. The framework hides all these details from the programmer, who therefore can abstract from these details and is able to concentrate on the functional aspects of his/her application.

We implemented our CMS framework in two environments: Java [12] and Oberon [18]. All examples and references in this paper relate to the Java implementation (JavaCMS). This platform allows programmers to 'play' with semantics for message transmission. It allows arbitrary new semantics or the composition of existing semantic actions into a new semantic, e.g. one could combine, asynchronous invocation with an infinite retry mechanism and a semantic that ensures some synchronization constraints. Every method of a remotely accessed object can have its own especially tailored semantic. The framework allows even alias like access to its objects, where every alias can have other associated semantics, i.e. one can use different semantics depending on the client that accesses the object.

The main restrictions of the JavaCMS framework – in regard to mobile computing – are the requirement for Java on both client-side and server-side, as well as the fixation of the client to interact with the network through the method invocation paradigm. However, we deem these restrictions as minor. Mainly because the method invocation paradigm does not serve as the true communication mechanism, but as the interface metaphor that simplifies the design and the implementation of client-side applications.

The paper is separated into several sections. Section 1 introduces the JavaCMS system with the help of a generic example and lays a solid ground by defining a formal background. Additionally, this section demonstrates some possible applications of the CMS framework as a platform for mobile computing. Section 2 presents a possible formal specification of time-independent invocation, Section 3 explains the implementation of TII. Finally, Section 4 and 5 show some measurements and conclude the paper.

1. The JavaCMS System

In this section, we describe our basic framework JavaCMS using two different approaches. First, we give an informal introduction with the help of an extended example. Second, we show a small example to demonstrate the uses of JavaCMS as a platform for mobile computing. Finally, we try to give a formal description of the framework with the help of evolving algebras.

1.1. Putting JavaCMS to Work

This section gives an overview of our library using the 'Dining Philosophers' problem [6] as an example. This problem is well suited to show the advantages of our framework.

The message semantics of common object-oriented environments are fixed. The system either enforces one fixed semantic, or allows the choice between a small fixed set of semantics, each associated with some pre-defined keywords. Our invocation abstractions offer an open way to create arbitrarily many new kinds of semantics. For every method one can define the semantic that handles the invocation of that method: this is done by creating an instance of the invocation class and assigning it to the desired method. While doing so, two semantics must be supplied: caller-side (client) and callee-side (server) invocation semantics (see *Fig. 1*).

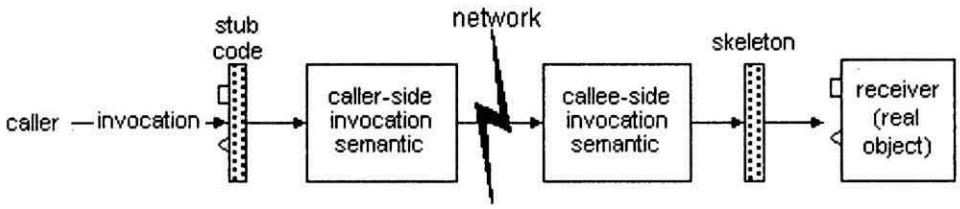

Fig. 1 Layout of intercepted invocations

The chosen client-side semantic is executed on the host of the stub object while the corresponding server-side semantic is executed on the host of the real object. This distinction has two advantages. First, the programmer can decide, individually for each part of the invocation semantic, where it should be executed, i.e., on the client or on the server. Second, when several hosts have a stub of the same server object, a client-based modification is executed only when the corresponding stub object is invoked. A server side modification is executed whenever a method is invoked on the real server object, i.e., regardless of the stub that initiated the invocation.

To introduce our implementation of the dining philosophers (see *Fig. 2*) we first present a straightforward RMI implementation that ignores all synchronization concerns and which is – of course – not correct.

Interface

```
interface Philosopher extends Remote {
      void think ();
      void eat ();
}
```

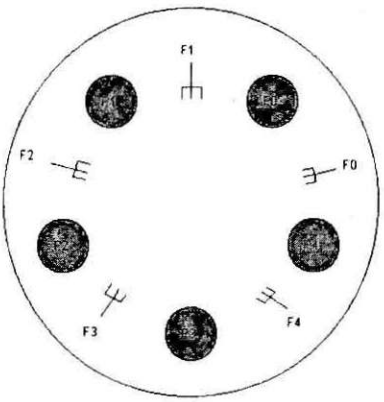

Fig. 2 Table layout for five philosophers

Server

```
class PhilImpl extends UnicastRemoteObject implements Philosopher {
   static Fork forks[] = new Fork[5];
   int left, right; // index of left and right fork
   static {
      for(int i=0; i<5; i++) forks[i] = new Fork ();
   }
   PhilImpl () throws RemoteException {}

   void think () {...}
   void eat () {
         ...
   }

   public static void main(String args[]) throws Exception {
      for(int i=0; i<5; i++) {
         PhilImpl p = new PhilImpl();
         if (i==4) {
            p.left=0; p.right=4;
         } else {
            p.left=i; p.right=i+1;
         }
         Naming.bind("Philosopher"+i, p);
      }
   }
}
```

Client

```
public class Client {
   public static void main (String args[]) throws Exception {
      Philosopher p = (Philosopher)
      Naming.lookup("//127.0.0.1/Philosopher"+num);
      for(;;) {
         p.think();
         p.eat();
      }
   }
}
```

To correct this faulty behavior we have to insert a synchronization code. The straightforward approach is to protect the invocation of *eat* by declaring it as a synchronized method. Unfortunately, it is not possible to use the *synchronized* keyword for the declaration of *eat*, as it synchronizes on the receiver of the message *eat*. However, we need to synchronize on the receiver's two forks. Therefore, we need to use another kind of synchronization: *synchronized* blocks. *Synchronized* blocks intermix application (functional) code with code responsible to guarantee synchronization constraints (non-functional requirement) and requires us to redesign the method *eat*.

```
void eat () {
    synchronized (forks[left]) {
      synchronized (forks[right]) {
      ...
      }
   }
}
```

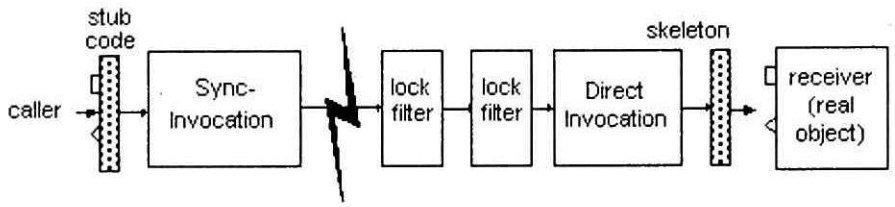

Fig. 3 Semantic for *eat* using JavaCMS

Using our composable message semantics (CMS) we can avoid this mixture. We use JavaCMS to modify the invocation semantic of the method *eat* (see *Fig. 3*).

Server

```
public class PhilImpl {
   void think () {...}
   void eat () {
      ...
   }
   public static void main (String args[]) throws Exception {
      Fork forks[] = new Fork[5];
      int left, right; // index of left and right fork
      int i;
      for(i=0; i<5; i++)
         forks[i] = new Fork();
      for(i=0; i<5; i++) {
         PhilImpl p = new PhilImpl();
         ClassInfo ci = new ClassInfo(p);
         CalleeInvocation inv = new DirectInvocation();
         if (i==4) {
            left=0; right=4;
         } else {
            left=i; right=i+1;
         }
         inv = new SynchronizedInvocation(inv, forks[left]);
         inv = new SynchronizedInvocation(inv, forks[right]);
         ci.getMethod("eat").setCalleeInvocation(inv);
         Remote.export("Philosopher"+i, p, ci);
      }
   }
}
```

Client

```
public class Client {
   public static void main (String args[]) throws Exception {
      InetAddress server = InetAddress.getByName("127.0.0.1");
      int num = ...; // number of desired philosopher
      PhilImpl p = (PhilImpl)Remote.get(server, "Philosopher"+num);
      for(;;) {
         p.think();
         p.eat();
      }
   }
}
```

On the server, we first initialize the necessary forks, i.e. *forks*. Afterwards, we use a loop to initialize our philosophers. The *ClassInfo* constructor returns a *Class* object for the passed object instance. This *Class* object contains information about all the object's methods (including inherited ones). In particular, it contains

the necessary information to change the invocation semantics. We assign the new callee-side invocation semantic *inv* to the method *eat*; afterwards, we export the philosopher using the assigned semantic information *ci* by calling *Remote.export*.

In *inv* we define the callee-side semantic to be used for the method eat of the different philosophers. The semantic consist of two locking filters (*Synchronized-Invocation*) and the invocation abstraction *DirectInvocation* (see *Fig. 3*), which is part of the CMS framework and actually invokes the method. A locking filter first acquires its assigned resource (a fork in this example) and then passes the invocation on. The above example shows the separation of functional and non-functional code. The code necessary for the synchronization is concentrated within the initialization part. The actual application code stays as if there were no synchronization constraints.

In all five passes through this loop we create new instances of the locking filters with different associated objects (forks). This results in different semantics of the eat method for different philosophers (see *Fig. 4*). All semantics use two synchronization filters but synchronize on different objects.

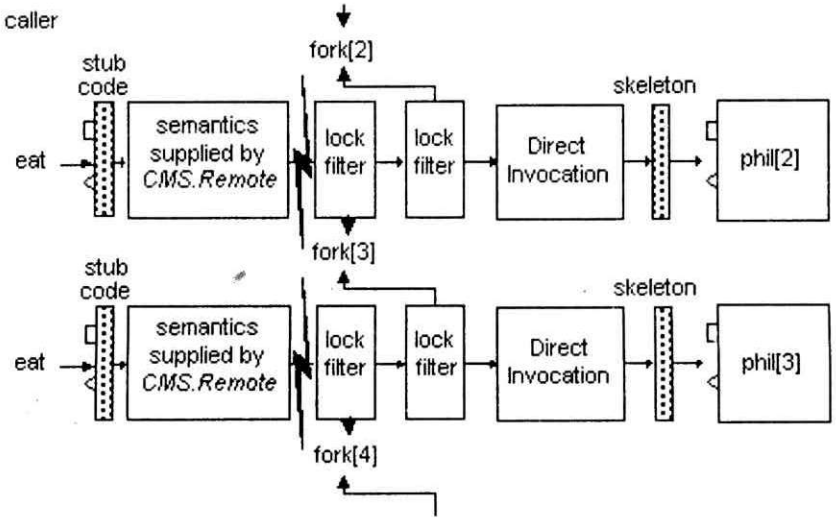

Fig. 4 Semantics for *eat* of different philosophers

The synchronization filter itself is a Java class that extends the abstract server-side filter class and overrides the method invoke. We introduced the filter for the example but it can be used by other arbitrary applications that need to synchronize on a specific object.

```
public class SynchronizedInvocation extends CalleeFilterInvocation
implements java.io.Serializable {
   private Object lock;
```

```
   public SynchronizedInvocation (CalleeInvocation inv,
   Object lockObj) {
      super(inv); lock = lockObj;
   }
   public CMSOutputStream invoke (Object obj, int id,
   CMSInputStream s) throws Exception {
      synchronized(lock) {
         return super.invoke(obj, id, s);
      }
   }
}
```

The above filter *SynchronizedInvocation* demonstrates how a programmer can add arbitrary new semantic actions by writing a new filter. The generic layout of an invocation filter is shown in the following listing:

```
public class MyFilter extends CalleeInvocationFilter {
   public CMSOutputStream invoke (Object obj, int id,
   CMSInputStream s) throws Exception {
         CMSOutputStream result;
         somePreprocessing(obj, id, s);
         result = super.invoke(obj, id, s);
         somePostprocessing(obj, id, s, result);
      return result;
   }
}
```

Invoke gets the receiver object *obj*, an identifier *id* that denotes the invoked method and a stream *s* that contains the marshaled parameters. As a return value, it supplies the stream containing the marshaled result of the invocation. Before and after the invocation is forwarded to the next abstraction, the filter can do its specific work. With the help of metaprogramming facilities, it can even scan the parameter stream and react to its contents.

In most applications it is not necessary to distinguish between callee-side and caller-side filters and abstractions. However, if one has to access the invocation's actual parameters the distinction is mandatory. The above example *MyFilter* is a callee-side invocation filter as it receives a *CMSInputStream* as its parameter. The skeleton code will read this stream in order to reconstruct the passed parameter. Finally, the filter passes back a *CMSOutputStream* into which the skeleton wrote the result value of the invocation. On the other side, a caller-side invocation filter receives a *CMSOutputStream* that contains the actual parameters as written by the stub and returns a *CMSInputStream* that can be read by the stub in order to reconstruct the result value of the invocation. Therefore, we currently have two abstract filter classes: *CalleeInvocationFilter* and *CallerInvocationFilter*. This distinction allows that the network layer starts to transmit the marshaled data before all parameters are completely marshaled. If one abandons this speed-up, it is sufficient to have only one filter class: *InvocationFilter*.

To write a new invocation abstraction, one has to decide whether a server or client side abstraction is actually desired. Similar to a filter, this requires declaring a new type that extends *Invocation*. The method invoke has to be overridden. However, unlike with a filter, it is not possible for an abstraction to handle the invocation with the help of a super call, i.e. it has to handle the invocation itself. The actions to achieve this depend completely on the goal of the new abstraction. For example, implementing delayed (time independent) invocation needs some kinds of storage area where the invocation information (*obj, id, s*) is stored for later retrieval. Additionally, it needs a mechanism that extracts invocations from this storage at a suitable moment and starts the actual invocation.

New server-side abstractions are easier, because the actual invocation is handled by our run-time system (*DirectInvocation*). It makes a synchronous method invocation. One will probably not replace it with another abstraction. However, decorating it with filters is possible and even desirable.

1.2. Using JavaCMS as a Platform for Mobile Computing

The JavaCMS system itself does not contain components supporting mobile communication [8]. In mobile environment the system assumes the existence of a communication facility like Mobile-IP [17, 16] or DHCP [2], with which the routing of network packets is guaranteed. However, because of its flexibility the system enables the programmer to implement invocation semantics with which CMS-based applications can collaborate in mobile environment.

Time-independent invocation (TII) could be one of the most important invocation semantics in mobile environment. This semantics has been defined as a new feature in CORBA Messaging Specification [14] by Object Management Group (OMG), and will be a part of the CORBA3 specification. Although our starting point was the OMG specification, our TII implementation does not follow it closely. We have adopted the notion of time-independent invocation to our system and implemented the most important features of the specification.

The following short example demonstrates the features and capabilities of TII in JavaCMS. First of all, we wanted to facilitate the communication of periodically unconnected mobile clients, thus we have implemented TII as a client-side semantics (*TIInvocation*):

```
long maxRetry = 100; // number of retries
long retrySleep = 1000; // idle time
CB cb = new CB(); // callback object for receiving responses
TIInvocation tii = new TIInvocation(maxRetry, retrySleep, cb);
ci.getMethod("mi").setCallerInvocation(tii);
```

In case of communication failure the semantics try to send the invocation request later, but the number of successive retries can be limited if necessary. Between two retries the semantics idle a given period of time in milliseconds. These two parameters must be supplied by the user as the first (*maxRetry*) and second

(*retrySleep*) parameter of the constructor. If the first parameter is set to zero, the semantics do not limit the number of retries. The third parameter is optional. This is a *callback* object which has to be given, if we want to be informed about the execution of the remote method, and we want to receive its return value. The callback classes must implement the *CallBack* interface:

```
public class CB implements CMS.Remote.CallBack,
java.io.Serializable {
   public CB() {}

   public void cb(CMSInputStream in) {
      try {
         System.out.println(in.readInt()); // unmarshal
         the return value (int)
      } catch (java.io.IOException ex) {}
   }
   public void exception(Exception ex) {}
};
```

As soon as the system receives the response message of the remote method, it calls the *cb* method of the callback object with *CMSInputStream* containing the return value. In this method we can unmarshal the return value and make other necessary computations. The system calls the method *exception* if it fails to deliver the invocation request or receive the results.

The below client-side code-fragment demonstrates the use of remote methods with TII semantics:

```
TT x; // the test object with method public int mi(int);
...
x = (TT)Remote.get(adr, "TT"); // import
...
try {
  System.out.println(x.mi(200));
} catch(CMS.CMSException e) {}
```

We can see, that invoking remote methods with TII semantics results *CMS.CMSException*. This is because JavaCMS unmarshals response messages inside the stub and the TII semantics leave it to the programmer. Thus, the semantics return to the application right after the invocation request issued with a stream containing no valid return values.

1.3. Description of JavaCMS using Evolving Algebras

We provide an abstract formal description of the basic parts of the JavaCMS framework using *evolving algebras*. Since the discovery of evolving algebras in 1988 [4], it has become clear that it allows the development of powerful and elegant specifications and descriptions of heterogeneous distributed systems.

The $IFACE$, $CLASS$ and OBJ sets contain the interfaces, classes and objects of a stand-alone CMS system. Each class has an interface which is implemented by the given class and each object is an instance of its class. The $iface : CLASS \rightarrow IFACE$ function provides us with the interface of a class and the $class : OBJ \rightarrow CLASS$ function gives us the class of an object. The $class$ function must be updated whenever an object is constructed via the $new(Class)$ operator

extend OBJ **by** Obj **with** $class(Obj) := Class$ **endextend**

or destroyed by the JVM

discard Obj **from** OBJ.

As our framework creates skeletons and stubs at run-time, it is possible to add arbitrary new elements to the $IFACE$ and $CLASS$ sets. These newly created classes (and interfaces) are generated using the meta-information of the exported objects that is stored in special *class information* objects of the class $ClassInfo \in CLASS$. The class information of a given object is obtained by the $class\_info : OBJ \rightarrow CI$ function. These $ci \in CI \subseteq OBJ$ objects hold the client and server-side invocation semantics that the user previously assigned to the given method. The client and server-side invocation semantics of a method are stored in a $mi \in MI \subseteq OBJ$ *method information* object. One can get the client-side semantic by the $caller\_sem : MI \rightarrow CSEM$ function, the server-side semantic by the $callee\_sem : MI \rightarrow SSEM$ function. The $method\_info : CI \rightarrow 2^{MI}$ function provides the set of method information holding the invocation semantics of all the methods of its object.

The $CSEM \subseteq OBJ$ and $SSEM \subseteq OBJ$ sets hold the client and server-side invocation semantics. These objects are instances of the descendants of $CallerInvocation, CalleeInvocation \in CLASS$ classes.

Each stub class is determined by the class information (CI), its network address ($NADR$) and the name (STR) of the corresponding remote object: $stub : CI \times NADR \times STR \rightarrow CLASS$. The STR set is an infinite set which contains all strings that conform with the Java class *String*.

Now, we can refine the definition of the interfaces by the method information objects. An interface is a set of methods (or method signatures), and each interface can be determined by the proper method set. This fact implies that all $if \in IFACE$ interfaces are defined as a subset of MI and $IFACE$ consist of a proper subset of MI.

As we have seen in the previous subsection, the $Remote.get()$ method must be used when we would like to obtain the reference of a remote object. As a matter of fact, this method does not return a reference of the given object but generates a stub class and returns a reference to an instance of this class.

In our formal model, objects communicate through messages. We model these messages by introducing the external function *event*, which for some given object returns the message it just received. We introduce the *self* object, which refers

the object that contains the described method, as well as the JVM object which represents the Java Virtual Machine. Furthermore, we assume that *event(object)* becomes *undef*, as soon as the object has read the current value.

Additionally, the formal model has to deal with different CMS systems, which sometimes exchange messages. We model these inter-CMS communications by the *forward <event> to Obj* and *return <event> to Obj* abstract updates. They update the *event* function of the remote object *Obj* with the corresponding events. As the remote system is identified by its network address, we had to introduce the $cms : ADR \rightarrow OBJ$ function, which yields the *Server* object of the remote system.

CMS.Remote.Remote::get()

if $event(self) = get(Adr, Name)$ *from* $User$
thenif $cms(Adr) \neq undef$
then
forward $resolve(Name)$ *to* $cms(Adr)$

if $event(self) = resolved(Ci)$
thenif $Ci \neq undef$
then
extend $CLASS$ **by** Sc **with**
$stub(Ci, Adr, Name) := Sc$
extend OBJ **by** $Stub$ **with**
$class(Stub) := Sc$
return $Stub$ *to* $User$
endextend
endextend

As we presented in the previous subsection, remote objects must be exported by the *Remote.export()* method. Whenever one exports an object, the system registers the object together with its class information. The $ROBJ \subseteq OBJ$ set holds the exported objects, the $INFO \subseteq CI$ holds their descriptor objects. As each exported object must have a unique name we can refer to the exported objects through their name. The $robj : STR \rightarrow ROBJ$ and $info : STR \rightarrow INFO$ injective functions yield the *object* and the *class information* corresponding to a given name.

CMS.Remote.Remote::export()

if $event(self) = export(Name, Obj, Ci)$ *from* $User$
then
extend $ROBJ$ **by** Obj **with**
$robj(Name) := Obj$
endextend
extend $INFO$ **by** Ci **with**
$info(Name) := Ci$
endextend

Whenever a client tries to obtain the reference of a remote object, it forwards a *resolve(Name)* event to a CMS server object. This object represents the remote

CMS system to its clients. The server object resolves the name by the $info$ and $robj$ functions and returns the result to the client object. The $fetch : CI \rightarrow \{true, false\}$ function determines whether the server has to fetch the object's fields, or not.

CMS.Remote.Server::run()

if $event(self) = resolve(Name)$ *from* $Client$
thenif $fetch(info(Name)) = true$
  then
    return $resolved(info(Name), robj(Name))$ *to* $Client$
  else
    return $resolved(info(Name), undef)$ *to* $Client$

One usually constructs a new ci class information from another object. Therefore, we have to extend the OBJ and CI sets and most of their functions. Each method will have a default client and server-side invocation semantic: $StdInvocation$ and $DirectInvocation$.

new CMS.Invocation::ClassInfo()

if $event(JVM) = new(CMS.Invocation.ClassInfo(OObj))$ *from* $User$
then
  let $MiSet = iface(class(OObj))$
  extend OBJ **by** Obj **with**
    $class(Obj) := ClassInfo$
    extend CI **by** Ci **with**
      $class\_info(Obj) := Ci$
      $method\_info(Ci) := MiSet$
      endextend
    endextend
  $\forall mi \in MiSet$: **extend** MI **by** mi **with**
    $caller\_sem(mi) := StdInvocation$
    $callee\_sem(mi) := DirectInvocation$
  endextend

The $setCallerInvocation()$ and $setCalleeInvocation()$ methods simply update the $caller\_sem$ and $callee\_sem$ function.

CMS.Invocation.MethodInfo::setCallerInvocation()

if $event(self) = setCallerInvocation(Sem)$ *from* $User$
then
  extend MI **by** Mi **with**
    $caller\_sem(Mi) := Sem$
  endextend
  where
    $Mi \equiv self$

CMS.Invocation.MethodInfo::setCalleeInvocation()

if $event(self) = setCalleeInvocation(Sem)$ *from* $User$
then
  extend MI **by** Mi **with**
    $callee\_sem(Mi) := Sem$
  endextend
  where
    $Mi \equiv self$

Even though JavaCMS is an object-oriented distributed objects system, our formal description deals with only one component of an application that has collaborating distributed CMS-based components. Therefore, we have to extend the above model to be able to describe the real distributed CMS system. Our *extended distributed model* comes as a finite set of evolving algebra programs which have separate sets and functions except the OBJ and $NADR$ sets, which we need to describe the inter-CMS communication. The separated algebras can be evaluated concurrently, except when an inter-CMS communication occurs and synchronization is needed.

The exact semantical definition of distributed evolving algebras is given in [5].

2. Specification of Invocation Semantics

In this section, we present a very simple description of various invocation semantics based on propositional temporal logics [11, 13, 10]. We do not intend to give a detailed description of the implemented semantics, but only try to introduce the basic properties that one expects by definition. These properties describe only the client-side invocation semantics. However, one could also give server-side descriptions.

An abstract program is represented by its finite action set P, which contains the operations of the given program. The execution order of these actions is restricted by the detailed program model.[1] The program P conforms to a specification, if the formulas are true to all possible[2] $\alpha \in (2^P)^{**}$ execution sequence.

The specifications below describe the effect of the semantics on the caller program and determine their communication properties. Thus, the atomic propositions are bound to the actions of a program and the basic communication steps. Let α denote the execution sequence that represents a possible run of the P program. According to α, the p atomic proposition in $j \in \{1..|\alpha|\}$ is true if, and only if $p \in \alpha[j]$. Generalizing this definition, we can get propositions about an arbitrary set of actions $S \subseteq P$. The S[3] proposition is true in j if, and only if $\exists s \in S : s \in \alpha[j]$.

[1] For example according to the sequential model, we can define a follower function which determines the next operation that is executed. This operation is determined by the last executed operation and the state in a state-space.

[2] According to the detailed program model.

[3] We will abuse notation and use $S \subseteq P$ sets both as an action set or a proposition.

Further generalizations are needed if the program consists of more than one thread of execution. We define a thread as a $T \subseteq P$ subset of the program, which is started and terminated by special actions. The actions of the ith ($i \in \mathbb{N}$) execution of the T thread are denoted by $t_i \in T_i$ ($t \in T$). The truth value of the atomic proposition $t_i \in T_i$ is determined as in the case of non-indexed propositions.

In an α execution sequence the actions of the thread T must be started by $begin.T$ and closed by $end.T$ actions. This implies that the first action that is executed is $begin.T$ and that the thread is terminated by $end.T$. A run of a thread is initiated by a caller thread action.

$$\forall i \in \mathbb{N} : \neg T_i unless\ begin.T_i$$
$$\forall i \in \mathbb{N} : \Box(end.T_i \rightarrow \Box\neg T_i))$$

The Env_T set denotes the actions of the caller thread. This set contains the action that initiates the execution of the T thread. The $begin.T$ and $end.T$ actions are contained within the thread itself:

$$\{begin.T, end.T\} \subseteq T$$

In our model, invocation semantics are thread bound by synchronization and communication properties. An invocation semantic S is initiated by the proper method invocation action $s.m$ where s is a remote object and m is one of its methods. The $s.m$ proposition is true in $j \in \{1...|\alpha|\} \iff s.m \in \alpha[j]$.

$$\Box(s.m \rightarrow \circ\exists j \in \mathbb{N} : begin.S_j)$$

The threads we introduced are inherently asynchronous. The execution of a thread does not influence the caller thread. It's very often necessary to synchronize the execution of the caller and the thread.

$$\forall i \in \mathbb{N} : \Box(begin.S_i \rightarrow (\neg Env\_S_i unless\ end.S_i))$$

In order to specify the communication-aspects of a semantic, we have to introduce four communication actions. With these actions the communicating parties can send and receive packets via the network. We leave the internal structure and behaviour of the network unspecified,[4] the only feature of the network we require is that it must be packet-switched.

The $req(j)$ action sends a packet with packet id $j \in \mathbb{N}$ to the server object. The $req(j)$ proposition is an atomic proposition. The req proposition is true in $k \in \{1...|\alpha|\} \iff \exists j \in \mathbb{N} : req(j)$. The $conf$ action reads a packet from the server object. The $conf(j)$ ($j \in \mathbb{N}$) atomic proposition is true in k if, and only if

[4]Without these specifications we can't tell anything about the network properties. It can lose and duplicate packets or change their order, but it could also guarantee safe communication.

the result of the $conf$ action in k is a packet, which has packet id j. [5] The $conf$ proposition is true in $k \iff \exists j \in \mathbb{N} : conf(j)$. The server object communicates with the client with $resp$ and ind actions. The definition of $resp$ is similar to the req action, the definition of ind is similar to the $conf$ action.

The below formula specifies a strict best-effort non fault-tolerant communication. However, it requires performing a communication action ($\Diamond req_i$). It also forbids retransmissions. The formula does not preclude the loss of confirmation messages. On the contrary, it precludes the repetition of the receiving action ($conf_i$).

$$\forall i \in \mathbb{N} : \Box(begin.S_i \rightarrow \Diamond req_i \wedge \Box(req_i \rightarrow \circ(\neg req_i \, unless \, end.S_i)) \wedge \\ \wedge \Box(conf_i \rightarrow \circ(\neg req_i \wedge \neg conf_i \, unless \, end.S_i)))$$

The standard *synchronous* semantic can be defined as a thread that synchronizes with its environment (caller thread) and conforms to the above formula. The standard *asynchronous* semantic does not synchronize with its caller thread, but still conforms to the above formula.

Oneway communication could be defined as a request-only communication, which does not include any confirmation. We can get a weaker specification, if we do not require the appearance of request events and simply prohibit confirmations.

$$\forall i \in \mathbb{N} : \Box(begin.S_i \rightarrow \Diamond req_i \wedge \Box \neg conf_i)$$

We define the *fault-tolerant push* semantic as an asynchronous, multi-response communication semantic. There must be a request message, which initiates an infinite number of confirmations:

$$\forall i \in \mathbb{N} : \Box(begin.S_i \rightarrow \exists j \in \mathbb{N} : \Diamond(req_i(j) \wedge \Box \Diamond conf_i(j)))$$

Time Independent Invocation is an asynchronous *fault-tolerant* semantic. The fault-tolerance implies, that there is a request-confirmation event pair with the same network message identifier and that the semantic cannot terminate until the valid confirmation action performs.

$$\forall i \in \mathbb{N} : \Box(begin.S_i \rightarrow \exists j \in \mathbb{N} : \Diamond(req_i(j) \wedge \Diamond conf_i(j)) \wedge \neg end.S_i \, until \, conf_i(j))$$

It is hard to satisfy this level of fault-tolerance, since the above specification expects that the client will be able to connect to the server object sometimes in the future. If the client will never be able to communicate to the server object, it will send infinite numbers of request messages, and this could result undesired congestion of the communication network and could exhaust processing resources. In the next section we will discuss this and other issues.

[5] $conf(j) \in \alpha[k] \iff$ the next packet we can read from the buffer of the network has a packet id j. We can give a more precise and formal definition only, if we define the internal structure and some behavioural aspects of the network.

3. Implementation of Time Independent Invocation

3.1. Receive Models

The client application can receive responses provided by a two-way invocation either by polling or through callbacks:

Polling model: In this model, each asynchronous two-way invocation returns a poller object. The client can use this object to check the status of the request and obtain the value of the reply from the server. If the server hasn't returned the reply yet, the client can elect to block awaiting its arrival or the client can return to the calling thread and check on the poller later when convenient.

Callback model: In this model, the client passes an object reference for a callback object as a parameter, when it assigns the (client-side) semantics to the method. When the server responds, the client system receives the response and dispatches it to the appropriate method on the callback object so the client can handle the reply.

In most cases, the callback model is more efficient than the polling model because the client need not poll for results and this continuous polling can cost certain amount of processing power.

It is easy to see that the callback and polling models are equivalent. The callbacks can be implemented by using the appropriate pollers, and the pollers can be implemented by using the appropriate callbacks.

3.2. Implementation Issues

The way one implements a new semantic considerably determines its flexibility and performance. In addition, the implemented semantic should be reusable, as all semantics are the building blocks of invocation abstractions/filters.

There are three ways for implementing a new semantic. According to the *stub-based* approach, the stub itself implements the semantics. This approach rejects the principle of general stubs, i.e. we have to deal with specialized stubs that are determined by one or more invocation semantics. Even though this approach could result in fairly efficient and flexible stub-code, there are also a number of drawbacks. First of all, the reusability of the semantic is quite restricted in invocation abstractions: the stub-based semantic must be on the first place of the filter. In the case of TII this restriction comes into conflict with the very essence of the semantic, as its fault-tolerant behaviour should be guaranteed on the end of a filter. Following this approach, we could face another problem. How do we implement more than one semantics in a single stub? If we allow the user to specify multi-semantics behaviour on the stub-code, we have to be able to compose the semantics inside the stub.

The *abstraction-based* approach leaves the stub-code untouched and implements the new semantic within a well-defined framework. However, even though this framework could restrict the developer and may result in additional overhead, the code is much more reusable than in the previous case.

The *utility-based* approach implements the semantic outside the given application by an invocation proxy-server. According to this approach the applications willing to invoke a remote method forward their requests to a persistent invocation server, which performs the remote method call using the defined semantic and stores its result (if any). In this case, the clients have to communicate with the invocation server via a secondary invocation semantic. Although this approach leads to the least efficient implementation and is the most complicated to setup, there is a great advantage using persistent invocation proxy-servers: we can perform persistent remote method invocations. Mobile clients can exploit this feature too, as they can be shut down and can move to another point of the network without losing the result of an invocation request.

We have applied the abstraction-based approach, as it is easy to implement and results in a highly reusable code. Moreover, the abstraction-based approach does not exclude persistence inside an implemented semantic. Therefore, we can extend the functionality of a semantic with this feature whenever it is necessary.

Finally, we had to decide which response model should be used. However, the two models are equivalent, because of its efficiency, we have chosen the callback model.

4. Performance Measurements

In this section we compare the performance of RMI (Remote Method Invocation) and our JavaCMS system. Although our system has not been optimized, our prototype enables us to draw interesting conclusions about the cost of our flexible approach.

As our test environment, we used a Pentium 233 MMX computer with 64M RAM running Debian GNU/Linux Potato (kernel version 2.2.14) and JDK-1.2. We measured the execution time of 10000 invocations 10 times, and calculated the average execution time. To have our measurements independent of the installed network and the current load, the client and the server were running on the same machine and communicated via the local TCP loop-back interface.

An instance of the following class *R* served as a remote object:

```
class R implements java.io.Serializable {
   public void M0() throws CMSException {};
   public int M1(int i) throws CMSException { return i; }
   public Obj M2(Obj r) throws CMSException { return r; }
}

class Obj implements java.io.Serializable { long l; }
```

We used three ways to communicate with the above server object: (1) Java RMI, (2) CMS with the default caller-side synchronous and callee-side direct invocation, (3) CMS with caller-side TII and callee-side direct invocation semantics.

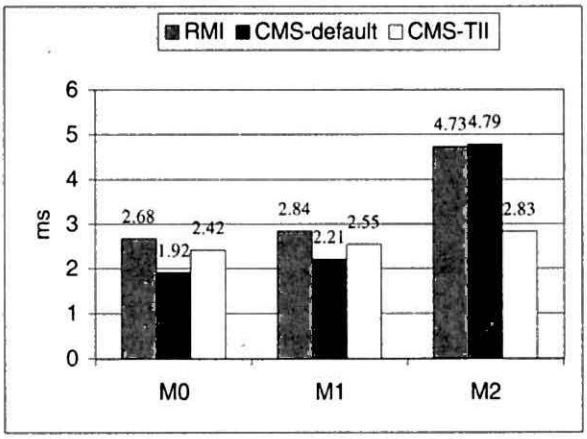

Fig. 5 Comparison of RMI and JavaCMS

Our results show (see *Fig. 5*), that although our flexible approach results in additional overhead, it took less time to invoke *M0* and *M1* with default CMS semantics than with RMI. Invoking *M2* via CMS took more time; however, RMI was only 1 percent faster. The performance of TII is quite stable as the network communication is done by a thread spawned by the semantics.[6] TII proved to be slower when we invoked *M0* or *M1*, because the cost of thread-creation outweighed the cost of network communication.

5. Conclusions

It is tempting to assume that all distributed interactions of a given application can be performed using just one (synchronous remote) method invocation abstraction, just like in a centralized system. In practice, this uniformity usually turns out to be restricting and penalizing and the myth of '*distributed transparency*' is very misleading. It is now relatively well accepted that the '*one size fits all*' principle does not apply to the context of distributed object interactions, especially in mobile environments, where in addition the communication paradigms are not fixed. Most *uniform* approaches to object-oriented distributed programming have recently considered extensions to their original model in order to offer a more flexible choice of interaction modes. For example, the OMG is in the process of standardizing a

[6]Further measurements show, that TII is even faster (relative to RMI and default-CMS) if we pass larger parameters.

messaging service to complement the original CORBA model with various asynchronous modes of interaction [14].

Several object-oriented languages offered, from scratch, various modes of communication. Each is typically identified by a keyword and corresponds to a well defined semantics. For example, the early ABCL language supported several keywords to express various forms of asynchrony, e.g., one-way invocation, asynchronous with future, etc [19]. Similarly, the KAROS language supported several keywords to attach various degrees of atomicity with invocations, e.g., nested transaction, independent transaction, etc. [3]. The major limitation of these approaches is that one can never predict the need of the developer, and coming with a new form of interaction means changing language.

We believe that the mode of distribution interaction should, like many other programming aspects, be represented by an extensible class library [11], and not be hard coded in the language. In other words, we advocate an approach where invocation modes are prompted the rank of first class abstractions. However, this approach emphasizes the importance of the exact (formal) specification of the programming framework and the invocation abstractions. The developer must be aware, that an extensible class library results not only in the freedom of development, but the responsibility of strict and precise specification.

We illustrated our approach by building a distributed extension to the Java language and we demonstrated it on a simple example. Moreover, we have given a simplified formal description of our system using *evolving algebras* and demonstrated the capabilities of JavaCMS extending it with *Time-independent invocation*, what we have specified formally using *temporal logics*. The very same approach could be applied to other languages and environments [7]. The actual requirements are easily fulfilled. The basic requirements are: (1) Run-time access to a compiler; (2) Dynamic code loading; and (3) Meta information.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, version 2.2, February 1998.

[2] DROMS, R. (editor), *Dynamic Host Configuration Protocol*, Internet Engineering Task Force, RFC 1531.

[3] GUERRAOUI, R.–CAPOBIANCHI, R.–LANUSSE, A.–ROUX, P., Nesting Actions through Asynchronous Message Passing: the ACS Protocol, In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, Springer Verlag (LNCS 615), 1992.

[4] GUREVICH, Y., Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pp. 1–57. Computer Science Press, 1988.

[5] GUREVICH, Y., Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[6] HOF, M., *Composable Message Semantics in Object-Oriented Programming Languages*, Trauner, ISBN 3 85487 118 X, 1999.

[7] HOF, M.–GUERRAOUI, R., *Abstracting Remote Invocations*, 1999.

[8] JING, J.–(SUMI) HELAL, A.–ELMAGARMID, A., Client-Server Computing in Mobile Environments, *ACM Computing Surveys*, **31**, No. 2, June 1999.

[9] KOZMA, L.–RÁCZ, É., A Specification Technique for Scheduling the Methods of Concurrent Objects, *Annales Univ. Sci. Budapest., Sect. Comp.* **17** (1998) pp. 253–268.
[10] KRÖGER, F., *Temporal Logic of Programs*, Springer Verlag, Berlin Heidelberg, 1987.
[11] LEA, D., *Concurrent Programming in Java*. Addison-Wesley, 1997.
[12] LINDHOLM, L.–YELLIN, F., *The Java Virtual Machine Specification* (2^{nd} Ed.) Addison-Wesley, April 1999.
[13] MANNA, Z.–PNUELLI, A., The Modal Logic of Programs, *LNCS 71*, pp. 385–409, 1979
[14] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05yes ed., May 1998.
[15] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, version: 2.2, February 1998.
[16] PERKINS, C., (editor), *IP Mobility Support*, Internet Engineering Task Force, Internet draft draft-ietf-mobileip-16, April 22 1996.
[17] SCHILIT, B. N.–ADAMS, N.–WANT, R., Context-Aware Computing Applications, *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, pp. 85–90, 1994.
[18] WIRTH, N., The Programming Language Oberon, *Software-Practice and Experience*, **18**, No. 7, July 1988.
[19] YONEZAWA, A.–TOKORO, M., (editors). *Object-Oriented Concurrent Programming*, MIT Press, 1987.