1-1-2008

An implementation of active objects in Java

George Oprean
University of Nevada, Las Vegas

AN IMPLEMENTATION OF ACTIVE OBJECTS IN JAVA

by

George Oprean

Bachelor of Science
Technical University, Cluj Napoca, Romania
2005

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science Department
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2008

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

APRIL 18, 2008

The Thesis prepared by

GEORGE OPREAN

Entitled

AN IMPLEMENTATION OF ACTIVE OBJECTS IN JAVA.

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE.

[illegible] 4/18/08
Examination Committee Chair

[illegible]
Dean of the Graduate College

[illegible]
Examination Committee Member

[illegible] 4/18/08
Examination Committee Member

[illegible] 04/18/08
Graduate College Faculty Representative

1017-53

ABSTACT

An Implementation of Active Objects in Java

by

George Oprean

Dr. Jan 'Matt' Pedersen, Examination Committee Chair
Assistant Professor
University of Nevada, Las Vegas

Active objects are a form of multitasking for computer systems. Active objects manipulate their own execution thread instead of using the execution thread of the object that created them. When a method is invoked on an active object, the call returns immediately and the caller continues execution. Thus, active objects can be utilized to develop parallel applications.

Active object model can be implemented in a number of different ways: with patterns, external libraries or extending the language. The solution proposed by this thesis is to implement active objects by extending the Java language with new keywords. We have modified Sun's open-source Java Compiler to accept the added keywords and to translate them into regular Java syntax.

An 'active' modifier was introduced to mark active objects; active objects can be created on remote machines and communication with them is done using RMI; active object's methods can be executed asynchronously and the results obtained later using a new 'waitfor' statement.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Within the last two decades object oriented programming (OOP) has become increasingly popular. Not least because of the introduction of the Java programming language by Sun in 1995. One of the basic ideas of object oriented programming is encapsulation of method and data; each object holds its own data as well as methods operating on the data.

However, objects are *passive*, that is, when a method is invoked on an object, the executing thread is that of the caller. The method is not executed by the object itself. Since many different threads of control can hold a reference to an object, many different threads can be invoking methods at the same time, possibly leaving the state of the object inconsistent. Using the *synchronized* keyword only provides a fake sense of security [18] as a method called from within a synchronized method can call methods in other objects that can cause a call back into the original object and modify the internal data.

Another problem with the *synchronized* keyword is the tendency to overuse it, which can easily lead to deadlock. The developer of a class does not know if his class will be used in a multi-treaded environment or in a single threaded environment. Just to be on the safe side, he might synchronize all the methods. The synchronized methods have a performance overhead, resulted from acquiring the lock for the object whose methods are invoked. It may happen that the methods are not synchronized and we need to use them in a multi-threaded environment. If we do not have access to class code, we have to create synchronized wrapper methods.

The heart of the problem is simply that an object is a passive structure that is being violated by various threads. If the object itself were in charge of executing its own methods in its own thread of control, then such unfortunate action can be avoided. Active objects are objects that control their own executing thread and do not reuse the thread of the object that created them. In order to invoke a method on an *active object* the object will have to accept the method invocation in the form of the parameters as well as the name of the method to be executed, and subsequently executing the code of the requested method in its own thread of control, thus excluding other threads from touching both its data and methods.

If active objects can execute methods asynchronously, then the active object paradigm can be used for parallel programming. During the time the active object executes a method, the caller can perform other computation in parallel.

Active Objects and the Real World

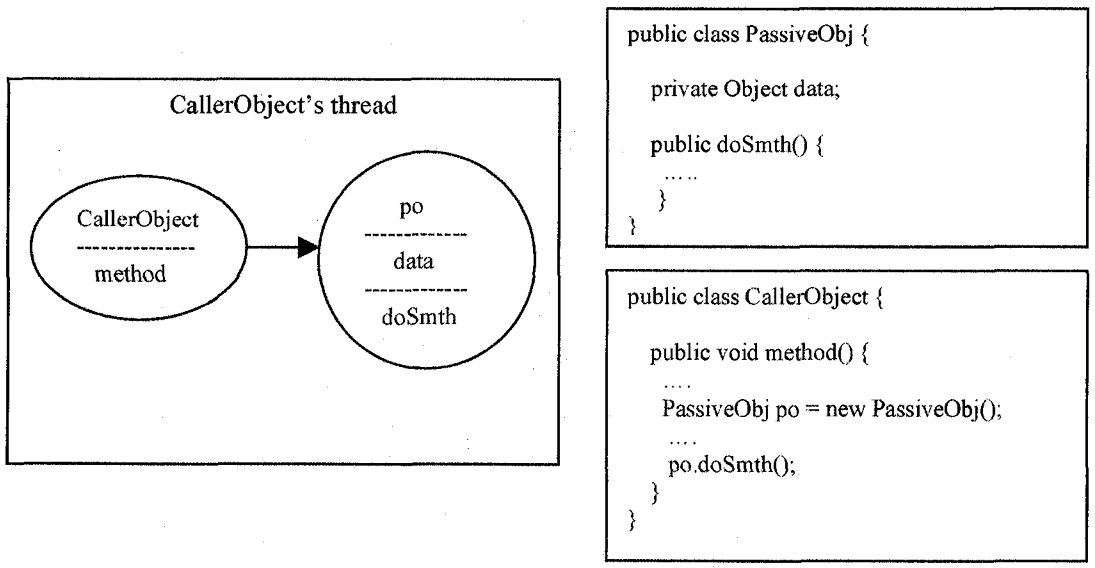
The argument that 'objects are considered harmful' has been borrowed from the process oriented design community. It is also argued that object oriented design is not a good reflection of how interaction between objects take place in the real world (which we model when creating software); the problem lies with the passiveness of the objects: If you are standing in front of your friend and want to borrow $20 you ask him, and he digs into his pocket and hands it to you. You do not ask and then reach into his pocket to retrieve the money. If that is the case, then why would we model systems this way? In standard OO, if you hold a reference to **friend**, and you wish to invoke the **borrowMoney()** method, then the call **friend.borrowMoney()** is executed in your own thread of control, not in a separate thread, thus breaking all similarity to the way the real world works. Interestingly enough, in OO, invoking a method is often referred to as 'sending a message'' (as in message passing, which in distributed computing means transferring data from one process to another through communication). This is *actually* what we *want*, but not what we do in a system with passive objects.

An active object receives messages (representing the requested method invocation) from the caller, performs the computation, and returns the result. An active object is comparable to the well known technique of Remote Procedure Call (RPC) or in OO terms Remote Method Invocation, which involve transferring method parameters and result back and forth between the calling process (client) and the remote object (server).

An active object system can be mimiced, and ultimately is implemented using RMI; As a matter of fact, a synchronous active object system can be easily used to implement a RMI system without stub generation.

Active Objects

In this thesis we implemented both asynchronous and synchronous communication with the use of active objects. Active objects are objects that control their own executing context and do not reuse the context of the object that created them. This is how *passive* objects work; they reuse the context of the objects that either created them or that invoke methods on the object. Active objects run in their own thread, different from the caller's thread.

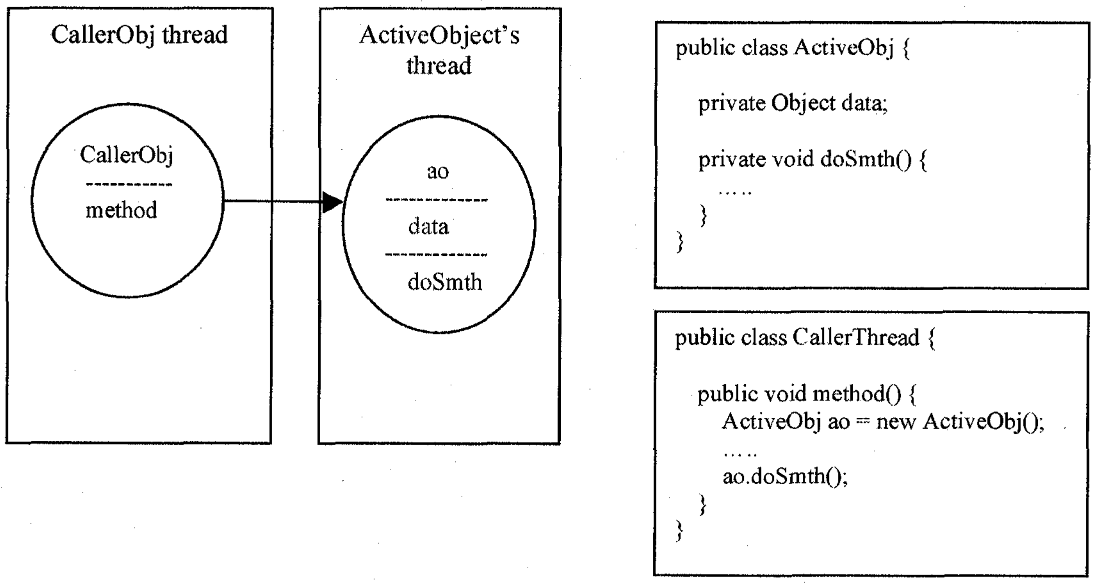


```
public class PassiveObj {
   private Object data;
   public doSmth() {
     .....
   }
}
```

```
public class CallerObject {
   public void method() {
     ....
     PassiveObj po = new PassiveObj();
     ....
     po.doSmth();
   }
}
```

Figure1: Passive object reuse the caller's thread.

An *active object* has its own execution context and can be used to model both synchronous and asynchronous communication. All the calls are executed on the active object asynchronously and the result of the invocation is obtained sometime in the future. The synchronous communication can be implemented by waiting for the result of method call right after the invocation. The client blocks immediately after the method invocation, thus resulting in a synchronous communication. The active object has a queue of pending requests and executes only one request at the time. If more clients have a reference to an active object and invoke methods on it, all the invocations are queued in the active object and it can decide what request to execute first. The active object schedules the requests to be executed based on: the order of arrival, the priority, or even the type of the object that invokes a method (for example a system object with *max* priority can be executed before a request from a regular object with the same priority).



```
public class ActiveObj {

   private Object data;

   private void doSmth() {
      .....
   }
}
```

```
public class CallerThread {

   public void method() {
      ActiveObj ao = new ActiveObj();
      .....
      ao.doSmth();
   }
}
```

Figure 2: Active object use its own execution thread.

By controlling its execution context, the active object will not 'be seen' in an inconsistent state, even if more than one object or thread has reference to it and access methods simultaneously. When programming using threads the objects that might be accessed simultaneously have to be written in a thread safe manner. The common way to do this is to use the synchronized keyword, but not even that assures that the object will not be seen in an inconsistent state [18]. Another problem is the tendency to overuse the synchronized keyword. This can lead to deadlocks because once the execution is inside a synchronized method or block (it acquired the lock for that object) then no other object can execute a method on the same object, until the lock for the object is released. The lock is released when leaving the synchronized block. So the following situation can occur. One object acquires a lock and calls a method on another object. But the lock for the second object is already acquired and is waiting for the first object lock to be release. This is a deadlock. No object can continue executing because they both are waiting for each other. Another problem with deadlocks is that they are difficult to discover and can very easily be introduced by modifying or adding code. In the example above, if some code is added and the second object does not acquire the lock for the first object, then deadlock does not occur. But unfortunately the reverse situation can happen too. The code runs perfectly, then we are modifying or refactoring some part of the code and this deadlocks the program.

Concurrent calls are resolved by the active object itself and only one method can execute at one time. Thus the object can not be seen in an inconsistent state. The active object paradigm is an alternative to doing parallel programming and can be extended to work in a distributed environment. Unfortunately, deadlocks can also occur with active objects.

Asynchronous Active Objects

As previously stated, our active object system can be used to implement any RMI system; no stub generation is needed as the system uses reflection and a general server manager to accept requests to create remote objects on remote machines. Though RMI does provide a way to improve execution time by executing code on machines that are perhaps better suited for a specific part of the computation task, it is not considered a typical technique for parallel computation; recall, a remote method invocation is, as a local method invocation, executed synchronously. This means that the caller blocks until the remote method invocation returns.

A well known technique for parallelizing computations is using message passing; the program is decomposed into a number of processes whose only way to synchronize is through communication using message passing. Messages must be explicitly sent and received. This can be done both synchronously and asynchronously.

The idea behind our asynchronous active objects is based on a merger between RMI and asynchronous message passing. An active object is placed on a remote machine when it is created. Any method invocation is done (automatically) though RMI (using a general client/server system and reflection), but such calls can be declared to be asynchronous. This means that the caller does not block and wait until the result of the method invocation is returned, but can continue executing immediately after. When the return value is needed, a new **`waitfor`** statement is executed. If the value had arrived before this statement is executed, the statement does not have any effect (will not block the caller's thread); if the return value is not yet present, the statement will pause the execution until the value is available.

This enables us to write parallel code that uses asynchronous active objects to achieve a speed up by using multiple processors. In a later chapter we will show an implementation of the Mandelbrot set computation using asynchronous active objects.

The organization of the following chapters is as follows. Chapter 2 presents other approaches to implement active objects and how these systems are different from or similar to our system. Chapter 3 gives a brief description of our system, the new keywords that were added, and the restriction on using them. The details of our system are presented in Chapter 4, including how the compiler was modified to accept the new keywords/syntax and how objects interact with each other to implement the active object paradigm. In the following chapters some examples of using our system are given, the results obtained are presented and conclusions are drawn. The last chapter presents the possibilities of extending or improving our system.

CHAPTER 2

RELATED WORK

The first time the concept of active object (actor) was presented was in to 1973 in Hewitt's actor model [6]. In his model, the actor was the central entity that executes its actions: communicates with other actors, creates actors and changes its behavior. The location of the actors can be distributed and they can execute actions in parallel. This actor (active object) model suits perfectly for creating distributed and parallel applications. Even though concurrent application can be developed using this model, other models of programming were more popular: procedural programming or Object Oriented (OO) programming. One reason is that during the late 80's and early 90's distributed or parallel programs were not a priority. It was enough if a program was running and gave the correct result. Memory space was much more important than execution time. Processor speed grew slower than memory space, so execution time became the priority. Execution time could sometime be reduced if the application were run on multiple computers/processors. The parallelism model offered by the procedural or OO model was not as powerful as the actor model, so a number of ways to integrate active objects into the OO languages have been proposed. Not only that, the active objects model supports concurrency, but it also can be used to develop distributed applications.

Remote Procedure Call

Distributed applications involve a high level of complexity: heterogeneous platforms, various network protocols, synchronous or asynchronous communication. One of the earliest approaches that tried to reduce this complexity was the Remote Procedure Call (RPC) [9]. RPC is a paradigm that implements the client/server model for distributed computing. The client makes a call to a remote object and then blocks while waiting for the response. The client resumes the execution when it gets the response from the server. The idea behind RPC is that local calls and remote calls should look the same to the programmer and thus he does not know if the server and the client are in the same address space or are distributed on different machines. The same principle has to hold for active and passive objects, that is, the calls should look the same to the programmer no matter where the objects are situated, local or remote.

Java Remote Method Invocation

Remote Method Invocation (RMI) [10] is the same as RPC for the Object Oriented world. There is no need to create a socket in order to communicate with a remote object when using RMI. All the details of communication are resolved by the RMI system. The location of an object is invisible to the programmer, so he interacts with remote objects just like with regular objects, by sending messages to them. The server (remote object) has methods that can be called remotely and these methods are defined in a remote interface. The remote class implements the remote interface and then an instance of the remote class is put into a registry, from where all the potential clients can access this instance. Using these remote methods is done just like a regular (local) object: the '.'

operator is used to access the methods. After compilation a stub and a skeleton is created. The stub is located on the client side and acts as a proxy for the remote object and the skeleton on the server. When a client calls a method on the remote object the invocation is sent to the stub which is responsible for marshaling the parameters. The skeleton unmarshals the parameters and sends the invocation to the actual remote object. Then the process is reversed: remote object→ skeleton→stub→client and the client obtains the return value of the invocation. The client does not have any knowledge of the location of the object, as all the communication between stub and skeleton happens behind the scenes.

Both RPC and RMI are synchronous client/server models. The client has to wait for the response from the server before it can continue executing the rest of the program. Some applications can be improved and better performance can be obtained if we can take advantage of asynchronous communication and utilize the 'waiting time' when the client is blocked until the result from the server arrives. During this 'waiting time', the client can perform other operations that do not involve using the return value from the remote invocation. Unfortunately, there is (normally) no support for asynchronous communication with RPC or RMI. If we consider a real world example, when somebody wants to prepare breakfast for his/her family and realizes that s/he does not have milk for the cereal, s/he can ask somebody (remote object) to get some cereal (result of the method call) from the store. During the time s/he gets the cereal (waiting for the result), s/he does not just wait and do nothing (this is what happens with synchronous communication). It is much more efficient to do some other activities related to preparing breakfast that has nothing to do with cereal: get the milk from the refrigerator, get the cereal bowls from the cabinets, set the spoons on the table and maybe pour orange juice in some glasses. Once s/he gets the cereal, breakfast can be served.

Message Passing Interface

In this thesis we implemented an active object paradigm having as a model the message passing model. Objects located on different machines communicate by exchanging messages. A well known implementation of the message passing model is the Message Passing Interface (MPI). MPI is a standard library used for developing parallel and distributed software by allowing processes to communicate with each other through message passing [11]. A process is started on each machine the program runs on. Before the processes start interacting with other processes though the MPI routines, the MPI environment has to be set up by calling the *mpi_init* routine. Also each process has to call *mpi_finalize* before it exits. This call cleans up all MPI state. MPI allows both synchronous and asynchronous communication and lets the programmer decide the type of interaction appropriate for each part of his code. If the programmer uses an asynchronous call, the execution continues immediately without waiting for the result. When he wants to use the return value of an asynchronous call, he has to stop and wait for the result. The program resumes once the asynchronous method is finished and the caller has the result of the invocation.

Many similarities exist between MPI and the way we implemented the active object paradigm, even though we did not implement it as a separate library, but by extending the language:

- There is an initialization routine that has to be run on each processor/machine before running the code.
- Both asynchronous and synchronous communication is possible and the programmer decides the type of interaction.
- Blocking the program until an asynchronous call (*send* or *receive* in MPI) finishes the execution.

Active Object Implementations

A number of different implementations of the active object paradigm exist:

- Employing patterns, Active Object pattern or Proxy Pattern.
- Extending the language by adding new keywords.
- Using external libraries.

We will briefly introduce some of these in this section.

Employing Patterns

Active Object pattern can be used to simulate active objects. This pattern decouples method execution from the method invocation to enhance concurrency. An example of how this pattern can be implemented is by using concepts/classes like *Proxy*, *Scheduler*, *Servant* or *Activation_Queue* [1]:

- *Proxy* represents the public interface of the active object (the methods available for other objects to call).
- *Scheduler* runs in a different thread than the client's, receives the calls, and manages a queue of requests deciding the order that the requests will be executed.
- *Activation Queue* stores the actual requests created by the Proxy.

- *Servant* represents the active object whose methods are available to the clients through the Proxy.
- *MethodRequest* objects are used to pass the context information about an invocation (parameters, method name). For every invocation of an active method, the Proxy creates a MethodRequest and sends it to the Scheduler to queue it.
- *Future* object used for allowing the clients to get the result of a method call after the Servant has finished executing the method.

There are some variants that reduce the number of components that have to be implemented; one variant is to integrate the role of the Proxy and of the Scheduler in just one component; another variant is to use message passing where the Proxy and the Servant are removed and the client exchanges messages directly with the Scheduler. In order to use active objects through this pattern, one has to first understand how all these components communicate with each other, and then to implement them. The programmer does not only have to concentrate on implementing the actual active object but also to code the components that are dealing with the interaction between the client and the server.

Another possibility of implementing active objects is through Java Dynamic Proxies [2]. While reflection allows the programmer to generate method calls at runtime, dynamic proxies facilitate the creation of an interface implementation at runtime. The dynamic proxies work by passing a callback object that will get invoked for any methods of that interface. In order to using this pattern, the programmer is not exposed to, or does not have to program all the details of client-server interaction. Moreover, there are fewer classes to be created than for the Active Object pattern. Understanding a simple example implemented with this pattern might take the programmer a fair amount of time. There has to be a more simple way to implementing active objects, because the concept itself is pretty straight forward: call an object asynchronously or synchronously, and at some point get the result back.

Extending the Language

One attempt to simplify the process of creating and using active objects for the programmer was made by Claude Petitpierre [3], by trying to integrate in the Java language new keywords and new syntax. He modified the Java language to integrate synchronous interaction with active objects. An active object contains a thread that is started when the object is created. Then any calls to a synchronous active object **A**, will have to block until **A** is ready to accept a call.

```
public active class A {
  public void method() {
    .....
  }
  public void run() {
    .......
    accept method;
    .....
  }
}
```

```
public active class B {
  public void run() {
    A = new A();
    ....
    a.method();
    .....
  }
}
```

Figure 3: Two classes implemented using the Petitpierre's extended Java language.

This approach uses the MPI model of sending and receiving synchronous messages. In the MPI model, two processes communicate by exchanging messages. The source of the message calls a *send* routine, and the receiver process calls a *receive* routine. If the communication is synchronous, the source process is blocked until the receiver process gets to the *receive* routine. Similar if the receiver process gets to the *receive* statement, before the source process actually sent the message, the receiver blocks. In Figure 3, when **A** gets to the accept statement, it is blocked until object **B** gets to call the method on object **A**. Likewise the method call on **A** in class **B** will block until object **A** gets to the rendezvous, that is to the accept statement.

Four keywords are added to the Java language: **active**, **accept**, **select** and **waituntil**. The **active** keyword is a modifier indicating an active class. The **accept** statement is used for synchronization, similar to a *receive* routine in MPI. The **select** statement is similar to the switch statement and gives the object the possibility of preparing several calls in parallel as the object does not know what event will occur first.

```
public active class A {
  public void doSmth() {
    ....
  }

  public void run() {
    .....
    select{
      case
         accept doSmth(); // accepts a call to doSmth() method
         doSomething1;
      case
         b.otherMethod(); //issues a call in another object. The call has to be accepted.
         doSomething2;
      case
         waituntil(2000);
         doSomething3
    }
  }
```

Figure 4: An example of select statement.

In this example, what ever event occurs first only the corresponding case will be executed and the other case statements will be dropped. Object **A** prepared both an acceptance to its **doSmth()** method(first case) and a call to another object (second case). Remember that for a call in an active object to be executed it has to be accepted in that active object. The call and the acceptance have to reach the rendezvous point before they can continue with execution. In this example, either the accept statement (first case) or the method call (second call) can reach the rendezvous point. Which ever reaches the rendezvous point first, the case corresponding to that event will be executed. If none of the two statements get to the rendezvous point in two seconds then the third case will be executed.

An issue with this approach is that by the time **A** gets to the select statement, both first and second case have their rendezvous partners ready and waiting. An object called the **doSmth()** method on **A** and now is waiting for the acceptance. Similar a **B** object executes the acceptance statement of its **otherMethod** (similar to *receive* in MPI) and now waits for another object to initiate this call (to execute a *send* routine in MPI). At this moment **A** can execute any of the first two cases. The selection decision is done randomly and this can lead to a nondeterministic behavior of the program.

When two objects get to the rendezvous point, that is one object invoked a method (similar to executing a *send* routine in MPI) and the object accepted the method (similar to executing a *receive* routine in MPI), the method invoked will be executed in the thread of the caller object. The method should be executed in the thread of the called object and only the result should be returned to the caller. Active object should execute the methods in their own execution context or thread and not in the thread of other objects, even if these objects are active too.

Java is not the only object oriented language to integrate active objects. Methods for expressing active objects have been developed for C++ by adding new keywords [2]. When defining a class, the **`active`** keyword is used to flag the active objects. This keyword can not only used in the class definition, but also in the inheritance definition. For example, if class **`A`** is a regular passive class, and it is extended by the active class **`B`**, using the **`active`** keyword in front of class **`A`** transforms it to an active class:

`active class B : active public A` { //declaration of class B }

Another keyword introduced is **`passive`**, used when an instance of an active class is to be utilized in a regular a conventional manner. For example,

`passive B obj_passive;`

where B is an active class, will invoke methods on **`obj_passive`** in the caller thread. The implementation of active objects has a transaction service to maintain the atomicity of client's invoking sequence. If a client declares a transaction for an active object, all the calls to that active object will be blocked until the transaction is over. The communication between objects is done both asynchronously and synchronously depending on the return type of the invoked method. The getter methods (methods that return data members) execute synchronously and invocations without a return value execute asynchronously. Invocations with return values are replaced with methods without return value at compile time and private data members are inserted to store the return value; methods to access this data (getters) are also generated.

Using External Libraries

The French National Institute for Research in Computer Science and Control has created a library (ProActive) for active objects [4] that can be integrated with the Java language (similar with the MPI library for C and Fortran). ProActive is a GRID middleware Java library for parallel, concurrent and distributed computing. The programmers simply write Java code and no modifications are needed to the JVM or to the compiler. There are two ways of creating an active object:

```
ProActiveObject.newActiveInParallel(..);

ProActiveObject.newActive(..);
```

as well as the possibility of transforming a regular passive object into an active one with:

```
ProActiveObject.turnActive(..);
```

Object type, passive or active, or object location, local or remote, are transparent to the programmer. When an asynchronous method is invoked, the call returns immediately and a future object is returned. The future object is used as a placeholder for the real return object. When a method is invoked on the future object and the return value is not available, the call blocks until it becomes available.

There are several restrictions when using the ProActive library:

- No final methods can be used in active objects.
- Final classes cannot be used to instantiate active object.
- There is the possibility of turning a regular object into an active one, but the programmer has to make sure that there are no other references to that object, because when other references are used for calling methods directly on the passive object, the model becomes inconsistent.
- The syntax of creating an active object is cumbersome and is different than the syntax for creating regular passive objects (the **new** keyword is not used for creating an active object).

- The programmer cannot control whether the calls are synchronous or asynchronous. The system tries to invoke all the methods asynchronously and whether a method can be invoked asynchronously is determined by the type of the caller or by the returning type.
- The use of future objects (that is the placeholders for the actual return objects in case of an asynchronous call) has drawbacks in the ProActive model; for example, not allowing the programmer to override functions like: *hashCode* or *equals*. Also trying to print a future object for debugging purposes using *toString* method can lead to a deadlock.

From Objects to Actors

Several active object models have been proposed; the first was Hewitt's actor model [6]. Later a mathematical definition for the behavior of an actor system was presented [7]. This model adopts the theory that everything is an actor, just like the Object Oriented paradigm considers everything an object. An actor is a computational agent that executes its own actions: initiate communication with other actors, create more actors and specify the replacement behavior. There is no ordering sequence of these actions and they can happen in parallel. Each actor has a mail address (target) that is used for specifying the destination of a message. Actors exchange messages which are stored by the system in buffers until they can be delivered to the target. The actor that sent the message can continue his actions immediately, without waiting a notification that the message was received. Thus, the interaction between actors is asynchronously.

All these characteristics from the actor system make a good concurrency model that can be extended to open distributed systems. The actor model is an improvement to the object oriented model and better suited for the development of next-generation distributed systems [8]. Object oriented languages, such as Java, should use objects as a unit of concurrency, just like actors are the unit of concurrency in the actor model. In Java concurrency is realized by using threads and only the low-level *synchronized* keyword is available for protecting a shared resource to be access simultaneously by multiple threads. Even lockes and mutexes must be implemented this way. When implementing an object the programmer does not know the context in which the object will be used, so the general tendency is to use the synchronized keyword for safe access to objects methods. This overuse of synchronized can easily lead to deadlocks. The actor model utilizes synchronizers, which are a linguistic abstraction that define the synchronization constraints over actors. The synchronizers allow the specification of a message patterns which are associated with rules to enable or disable methods on an actor. Synchronizer can be applied to a single actor or to a collection of actors and because they are not embedded in method definition they can be reused on different actors. Figure 5 shows a synchronizer applied to two managers, **adm1** and **adm2** that distribute resources to clients and there is a maximum limit of resources that can be allocated by both managers.

```
Allocation Policy(adm1,adm2, max) {
init prev = 0;

prev >= max diables (adm1.request and adm2.request),
(adm1.request xor adm2.request) updates prev : = prev + 1,
(adm1.release xor adm2.release) updates prev: = prev -1,
}
```

Figure 5: A synchronizer for allocation of resources.

When the maximum number of resources have been allocated the request method of both **adm1** and **adm2** objects have been disable. Otherwise, the resource is allocated and the number of resources already used is increased. Similarly, when a resource is released the total number of resources allocated is decreased.

Object Oriented languages allow only synchronous method invocations. Threads can be used to simulate asynchronous interaction but this requires the programmer to explicitly code this interaction since no support is offered by the language. In the actor model all the communication is done asynchronously by passing messages that are handled in a serialized fashion by a master thread:

```
a1: message1(args) a2:message2;
```

Actor **a1** sends an asynchronous message to **a2** and then resumes execution. Actor **a2** will reply to **a1** with another message, **message2**. Actors are thus a natural unit of concurrency and synchronization. In our active object system an object sends an asynchronous message (invokes a method) to an active object. The active object replies with a message (the return value) only if the method invoked does not return void. Our system also allows synchronous communication with the active objects. The activities executed by the active objects are similar to the activities of an actor: interact with other objects by exchanging messages, creating other objects and define a behavior (attributes of an object define the state and the methods define the behavior).

Our active object system implements the actor model, where messages are exchanged asynchronously. We chose to implement the system by extending the Java language with four new keywords: **`active`**, **`async`**, **`on`** and **`waitfor`**. The first keyword was also added by Petitpierre to his active object system. The **`active`** keyword can appear only in the class definition and marks all instances of that class as being active objects. Unlike other active object system, an active instance cannot be turn into a passive one or a passive object can not become active. The **`waitfor`** keyword is used as a barrier synchronization, where the caller thread has to wait for the result of the asynchronous invocation (Petitpierre uses the **`accept`** keyword for synchronization). The **`on`** keyword is used to extend the conventional creation of objects and not by calling a method that returns an active object (the Proactive way to create active instances). Once an active object is created, it can accept method invocations either from other active or passive objects. The communication between objects is realized through RMI and can be synchronous or asynchronous (**`async`** keyword makes an invocation asynchronous).

CHAPTER 3

ASYNCHRONOUS ACTIVE OBJCTES IN JAVA

We have implemented our asynchronous active object system in Java for a number of reasons:

- The language is already object oriented as we wanted to develop an active object system that would fit the Object Oriented model (for non-object oriented languages MPI can be used).
- It supports reflection. In order to create an object on another machine, we are only sending the name of the class and the constructor arguments to the host machine, and reflection is used to create the object.
- It has RMI built in. We are using RMI as the communication layer between objects situated on different machines. Of course at least one of the objects that participate in the interaction has to be active, otherwise all the communication happens locally and RMI is not needed. Also if a class definition is not available on the local machine RMI can dynamically download it.
- The Java compiler is available as open-source. Our implementation relies on extending the language by adding new keywords and modifying some of the syntax. Instead of creating the system for a subset of the Java language, since we then had to build a compiler from scratch, we used Sun's open-source Java

compiler and adapted it to parse the new syntax and generate the appropriate code.

- Autoboxing is done implicitly from Java version 1.5 (Sun's made its JDK 1.6 open source and we modified the compiler from that JDK [14]). One restriction with RMI is that the objects that are passed, either as an argument or as a return value, have to be serializable. This constraint eliminates the possibility of using primitive types with RMI. This is where autoboxing comes in handy. It transforms a primitive into its corresponding wrapper class (e.g. int to Integer).
- It is platform independent. Besides the parallelism obtained with the use of active object we also wanted our system to be distributed and to work on heterogeneous architectures without having to modify the application. Java supports this independency.

We define an **`asynchronous active Java object`** as an object that has the following characteristics:

- It must be **`active`**, that is, execute the methods in its own thread.
- It must be possible to place an active object on any machine reachable on the network that supports the Secure Shell (SSH) network protocol [12] and the Java Runtime Environment (JRE) [13]. The location of an active object is established when the object is created and its location can not be changed in time, so no mobility for the active objects. Although active objects can be passed as a parameter, that does not imply migrating the active object on another host. The active object stays on the machine where it was created and only a reference to it is sent as a parameter.

- Method invocation can be synchronous or asynchronous. Regular objects (passive) have all their methods executed synchronously, while an active object's functionality can be called either synchronously or asynchronously.
- A way to obtain the result of an asynchronous invocation must exist. When an active object method is called asynchronously and the method returns a value that is later used, the caller object/thread continues its execution immediately without waiting for the result. At some point, the caller's thread has to wait for the result of that call before it can execute the rest of the code.

New Java Keywords

In Chapter 2 we presented several approaches to designing an active object system: implementing patterns, external libraries or extending the language. We developed our active object system by adding new keywords to Java 1.6, and then modifying the Java compiler to parse and compile them. This addition consists of 4 new keywords/constructs:

- A new modifier **`active`**, which can only be placed on a class declaration. A class that contains the **`active`** keyword in its list of modifiers is called an active class. All instances of an active class are active objects.
- An extended object creation expression:

  ```
  new ActiveObj() on "machine_name";
  ```

  which creates an active object instance of type **`ActiveObj`** on machine **`machine_name`**. This new syntax can be used only with active classes. The **`on`** keyword can be omitted, but then the **machine_name** has to be omitted to. In this case the active object will be created on the current machine or localhost.

- An extended method invocation expression:

  ```
  activeObj.method(args) async;
  ```

  The **async** keyword makes the method invocation asynchronous, that is, the control returns immediately. For safety reasons an asynchronous method invocation may only appear on the right hand side of an assignment, or as an expression statement (i.e., where there is no return value, or where the return value is ignored). Methods of an active object can be invoked either asynchronously or synchronously. If the **async** keyword is missing, then the method will be executed synchronously and the caller's thread blocks until the invocation returns. The method will be executed asynchronously and a waitfor statement is added immediately after the invocation to get the result, thus obtaining a synchronous communication. The caller can continue execution only after he receives the result. This call will still be different than a regular, passive object method invocation, as the execution of the method happens in the active object thread and not the caller thread.

- A new blocking **waitfor** statement:

  ```
  waitfor activeObj variable;
  ```

  An asynchronous invocation on **activeObj** has been performed and now the result of this call has to be obtained. The **waitfor** statement causes the execution of the thread to be temporarily suspended until the asynchronous method invocation on **activeObj** has returned a value into the variable. If the method returns a value that is used later in the code, the thread must ensure that the return value is available. If this value is ready (the asynchronous invocation has finished) when the caller's code reaches the **waitfor** statement, then the variable will be assigned this value. Otherwise the thread will suspend its execution until the return value is available. If the method returns void, the **waitfor** is not necessary since no result is returned. If the method is invoked synchronously, the **waifor** is added by the compiler immediately after the call. A synchronous call is translated to an asynchronous call, immediately followed by a **waitfor** statement.

Restrictions of Using the New Keywords/Constructs

Our design of the active object system restricts the usage of the new keywords or constructs. This section will list these restrictions and gives a brief description on why the restrictions were imposed.

- The **active** modifier can only be used in class definition. Though a modifier, it can not appear in method, attribute or interface definition.
- The extended object creation expression can only be used when creating an active object. Passive objects use the regular syntax of creation, thus the following is not allowed:

  ```
  passive = new PassiveObject() on "machine1";
  ```

  All passive objects are created on the local machine, so no specification of the location is therefore needed.

- The keyword **async** can only be used after a method invocation and the object that the method is invoked on must be an active object. If method calls are chained one after another in the same statement and the **async** keyword follows, this only applies to the last method call:

  ```
  obj.foo().bar() async;
  ```

  The **obj** instance has a **foo** method that returns an active object and on that object the **bar** method is invoked asynchronously. The **foo** method on **obj** executes synchronously as **obj** is a regular passive object.

- Both the extended creation and the asynchronous method invocation can only appear on the left hand side of an expression and can not be passed as a parameter. The code will not compile and the compiler crashes. The following statement should be avoided:

  ```
  obj.method1(activeObj.foo() async);          (1)
  ```

  The **foo** method of the instance **activeObj** is called asynchronously and returns a value that is passed as a parameter to **method1**. But since the **foo** method is invoked asynchronously, the result is not available at that moment. So, we are trying to call a method with a parameter that will be obtained sometime in the future. The statement can be rewritten as follows:

  ```
  activeObj.foo() async;
  waitfor activeObj varName;
  obj.method1(varName);
  ```

- When invoking an asynchronous method the caller's thread will continue executing without waiting for the call to finish. If the method invoked returns a value, it is obtained through **waitfor** statement and assigned to an instance variable. So, when using asynchronous calls, the compiler enforces the programmer not to assign the return value using the `'='` operator. The following statement will cause the compiler to throw an error message:

```
value =  activeObj.foo() async;
```

and should be replaced by:

```
activeObj.foo() async;
waitfor activeObj value;
```

- The exact opposite happens when the method is invoked synchronously. The compiler enforces the programmer to have the return value assign to a variable using the `'='` operator:

  ```
      value = activeObj.foo();
  ```

  Recall that if the **async** keyword is not used, the invocation is executed synchronously, even on active objects. These restrictions are enforced by the compiler and the code will not compile.

- Waiting for the results of asynchronous invocation on an instance is done in the same order as the invocations occur:

  ```
  activeObj.foo() async;
  activeObj.bar() async;
  waitfor activeObj value1;
  waitfor activeObj value2;
  ```

value1 will have the return value of **foo** and **value2** will have the result of executing **bar** method and if the two types are not compatible a *ClassCastException* will be thrown during runtime (more on this in Chapter 4).

- Passing the value *null* as the argument of a method (or constructor) of an active object is not allowed. This restriction is the cost of using reflection when invoking a method on an active object (more on this in Chapter 4). When reflection is used to create an object, the class name and the arguments are needed and the constructor is determined based on the type of the arguments. But since null does not have a type, passing it as a parameter is not allowed. The same principle applies when passing null as a parameter of a method. If the following methods are part of an active object:

  ```
  public String foo(String s) {….}
  
  public Integer foo(Integer s) {…}
  ```

  and we invoke:

  ```
  activeObj.foo(null);
  ```

  there will be ambiguity in which method to call.

These restrictions are the cost of having a concurrency model build in the language. The Java language followed the threading approach to build parallel applications. When the programmer wants to create a concurrent application he has to create threads and each thread executes a task. The communication between threads and the access to shared resources can easily lead to deadlocks. These deadlocks are really hard to find and a simple code refactoring can create one. We tried to adjust the Object Oriented model to another model that is more appropriate to develop parallel and distributed applications, the actor model. We saw in Chapter 2, that even the approach of creating a library to extend the passive object model of Java to an active object model, enforced a lot of restrictions. Simply by adding another modifier to the class definition, **`active`**, and all the instances of that class can execute methods in parallel with other objects, either passive or active. Extending the creation expression of an active object by specifying the machine where the object will be located makes developing distributed application easier.

CHAPTER 4

IMPLEMENTATION

In this chapter we are going to present the details of our system and how we have implemented the active objects. The chapter is organized as follows: a general overview of the design is presented in the first section, and then the object/class that handles the communication on the client and server side will be detailed. The last part of the chapter shows the modifications to the Sun's Java compiler to accept the new keywords/concepts.

Design Overview

In the actor model, communication between actors is done by exchanging messages. This is also true in the object oriented model where an object sends a message to another object by invoking a method on the object (a method invocation is sometimes said to be a message being passed). The major difference is that in the OO model the communication happens synchronously. We wanted to implement active objects in Java so we followed the actor model: interaction between objects is done by sending messages and can be either synchronous or asynchronous, the active object executes the methods in its own thread and has a queue of pending messages that must be processed.

The program runs only on one machine and the code can create active objects on other machines. The machines that host the active objects don't have to run the whole program. This is different than the MPI approach, where the whole program had to be run on all the machines that participate in the execution of the program.

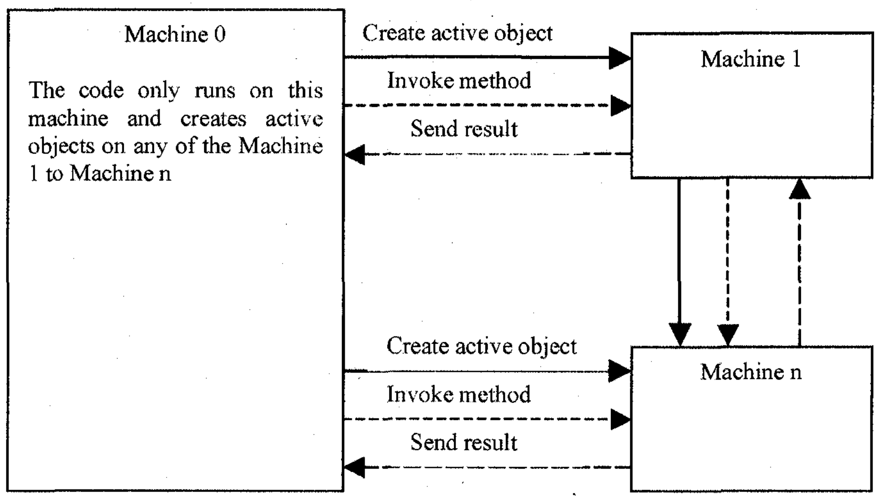


Figure 6: The execution model of our active object system.

In Figure 6, the main program runs only on one machine but it creates active objects on remote machines. An active object can in turn create other active object, either on the same or different machine.

The communication between active objects and regular objects (or between two active objects) is done by exchanging messages and is realized through RMI. We used RMI together with reflection to create an object or invoke a method on an object that resides on a machine different than the one where the program is running. Let us assume that the machine creating active objects or sending active object invocations is called *client* and the machine where the active objects are located are the *servers*. The client can send the server two kind of messages:

- **CreateMessage** asking the server to create a new active object
- **InvokeMessage** asking the server to execute a method one of its active objects

Creating an Active Object

The syntax for creating an active object on **server** is:

```
activeObj = new ActiveClass(args) on "server";
```

Assuming that this code is running on client, the client sends a **create message** to the server. The creation of the active object happens synchronously and the server replies to this message with another message sending the client a reference to the active object.

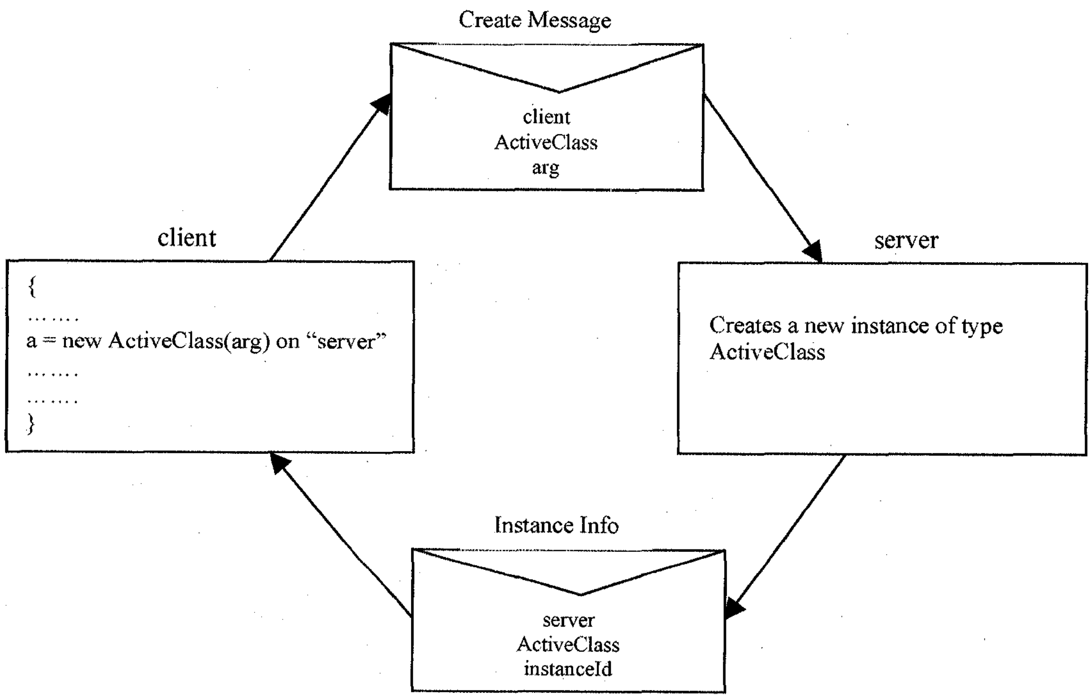


Figure 7: The process of creating an active object.

Figure 7 shows the messages exchanged by the client and the server when creating a new active object. The client creates a message that contains:

- The name of the machine where the call was initiated (as a String). In Figure 7 the creation of the new object happens on client. This is used internally by the server to keep track of invocations from each client since active objects can be passed around between clients (more on this later in the chapter).
- The type of the class and is passed as a String. The example in Figure 7, we are creating an object of type **`ActiveClass`**. The name of the class is used by the reflection mechanism to load the class definition.
- The arguments of the constructor. **`Arg`** is used to determine the constructor to be invoked for creating the class. **`Arg`** is an array of objects (type Object[]) that contains the arguments for the constructor.

Once the server receives the **`create message`**, it creates the object and stores it locally so it can be used for future invocations. The creation of a new object is done synchronously, so the client has to wait until the server creates the object. The server sends the client a 'remote reference' to the active object. It is not the physical object reference (which only has meaning on the server side), since the client and the server have different address spaces (even if the client and the server are situated on the same machine). The 'remote reference' is represented by an **`InstanceInfo`** object that contains the following information:

- The name of the machine where the object lives, in Figure 7 the machine name is server.
- The type of the active object, **`ActiveClass.`**

- An instance identifier to uniquely distinguish the instance from other active objects of the same class that reside on the same server.

The client then uses this reference every time it wants to call methods on the active object **a** (the compiler maps **a** to the remote reference sent by the server). The remote reference also contains the server name, so the client can determine the server it sends the request when invoking a method on **a**.

Invoking Methods on Active Object

Active object's method can be either invoke asynchronously or synchronously. Regular passive objects are invoked synchronously. Asynchronous invocation is what makes the active object different. Once the client has created the active object, **activeObj**, on the server, it can call methods asynchronously:

```
activeObj.foo(a,b,c) async;
```

The client sends the server an **invoke message**, asking the server to execute the **foo** method on the **activeObj** and return the result asynchronously. Recall, that at creation time, the client receives a 'remote reference' that identifies the active object on the server. When invoking a method on an active object the client uses this reference and passes it in the message. The server then knows on which instance it has to invoke the method.

Figure 8 shows the interaction process between the client and the server when invoking an asynchronous method. The client creates an **invoke message** that contains:

- The 'remote reference' of the active object. It is an object of type **InstanceInfo** that contains the information necessary to identify the object on the server side.
- The method name that has to be invoked, in this case **foo**, is passed as a String.
- The arguments of the method, **a**,**b** and **c**. The arguments are passed in an array of objects (e.g. **new Object[] {a,b,c}**).

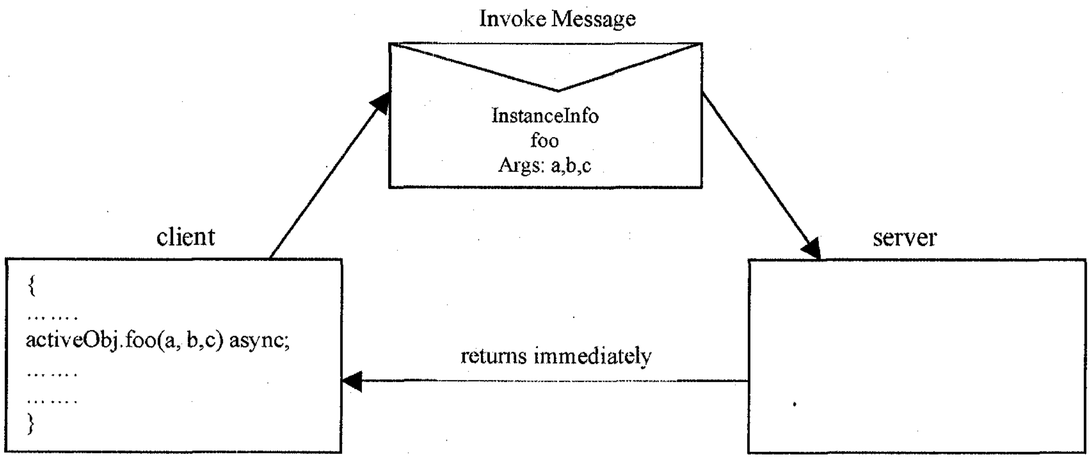


Figure 8: Invoking an asynchronous method.

The call returns immediately since the **async** keyword is used and the client continues executing the rest of the code. We have implemented this asynchronous invocation by creating a thread that performs the actual call. The thread gets the remote object for the server and invokes the method. Recall that RMI is used for interaction between the client and the server, so all the calls are synchronous. The thread just created has to wait for the RMI call to complete, but the main thread continues executing the rest of the code, obtaining the asynchronous behavior wanted. When the server finishes executing the method, the thread that initiated the call signals the main thread that the result is available. The main thread may be executing other methods and does not need the result when the thread received the result, so the result value is saved and the thread terminates.

If we invoke a method on an active object synchronously, the main thread has to wait for the result of the call unless the method returns void. In this case the invocation is sent to the server and the main thread continues its execution.

Waiting the Result of Asynchronous Invocation

When asynchronous interaction between the client and the server is used, the client has to get the result of the method invocation at some point. The **`waitfor`** statement is used for that:

```
…. o/c
activeObj.foo(a,b,c) async;
…..o/c
waitfor activeObj var;
….code that uses var (the return value of foo)
```

From the programmer's point of view, the **`waitfor`** statement can be translated as: "I'm waiting for the result of an asynchronous invocation on the active object, **`activeObj`**, and save the return value in the variable **`var`**."

In the above code, the **`foo`** method on **`activeObj`** is called asynchronously. The main thread continues with the execution up to the point where the return value of the **`foo`** method is needed. The programmer uses the **`waitfor`** statement to block the execution until the value is available. Recall that another thread, different than the main thread, carries out the asynchronous invocation **foo**. Let us call this thread **callingThread**, since it is the thread that actually does the invocation. Two situations can occur:

- The main thread gets to the **waitfor** statement **before** the **callingThread** finishes. In this case the main thread blocks and waits for the **callingThread** to send a signal when the value is ready. When the **foo** method has finished, the **callingThread** has the result and saves it in a shared resource and signals the main thread that value is available. The **waitfor** is thus a blocking statement.
- The main thread gets to the **waitfor** statement **after** the **callingThread** finishes. In this case the value is already saved in the shared resource where the main thread can get the value from and continue executing. Here the **waitfor** is non blocking statement.

In general a **waitfor** would be considered a blocking statement. The **waitfor** statements have to be used only for asynchronous methods that return a value. The order in which the asynchronous calls with return values are invoked on the same instance of active object has to be the same as the order in which waiting for the result is done:

```
activeObj.foo(a,b,c) async;
activeObj.bar(d,e)   async;
waitfor activeObj  varFoo; //the result value of foo
waitfor activeObj  varBar; //the result value of bar
```

The compiler will not throw any errors or warnings if the waiting order is reversed, that is, first wait for the **varBar** and then for **varFoo**. A *CastClassException* may be thrown at runtime if the return types of **foo** and **bar** are different and are not part of an inheritance hierarchy (i.e., the **foo** return type is String and **bar** return type is Integer).

The asynchronous invocation order on different active objects can be different than the order of waiting for the results:

```
activeObj.foo() async;
otherObj.bar() async;
waitfor otherObj varBar; //wait for the result of bar
waitfor activeObj varFoo; //wait for the result of foo
```

The programmer may forget to wait for the results of all the asynchronous invocations (or choose not to if the return value is never used). In this situation, when the method finishes executing its body, all the asynchronous calls initiated in the method body and the corresponding results that were not waited for will be discarded. When the thread that carries out the execution of an asynchronous call gets the result, it checks if the method body from where it was launched is still in scope. A method is still in scope if the end of the body was not reached or no return statement was executed. If the method is still executing, only then the thread saves the return value and signals the main thread. Otherwise the result is simply ignored.

This implies that all the asynchronous calls that were initiated in a method have to get the result in the same method (execute a **waitfor** statement). A method can not make an asynchronous invocation and then pass the active object to another method and then wait for the result in the second method. The following code is illegal:

```
Public class AClass {
    public void method1() {
        ActiveObj a = new ActiveClass() on "server";
        a.foo() async;
        int var = method2(a);


          }
         private int method2(ActiveClass a) {
         int var;
        waitfor a var;
        return var;
         }
    }
```

All the asynchronous invocations have a method identifier that corresponds to the method where the call originated. When waiting for an asynchronous result, the system looks only for at the asynchronous calls that were initiated from that method. Since no such call has occurred in **method2**, the system will throw and error saying that there is no asynchronous invocation for that object in the current method.

Message Ordering

Active objects can accept requests from any machine that has a reference to it. Active objects can be passed around, so not only the computer that created the active object can have a reference to it. The active object will respond to requests in the order received, on a first come first served basis. No ordering can be imposed on invocations from different machines on the same active objects.

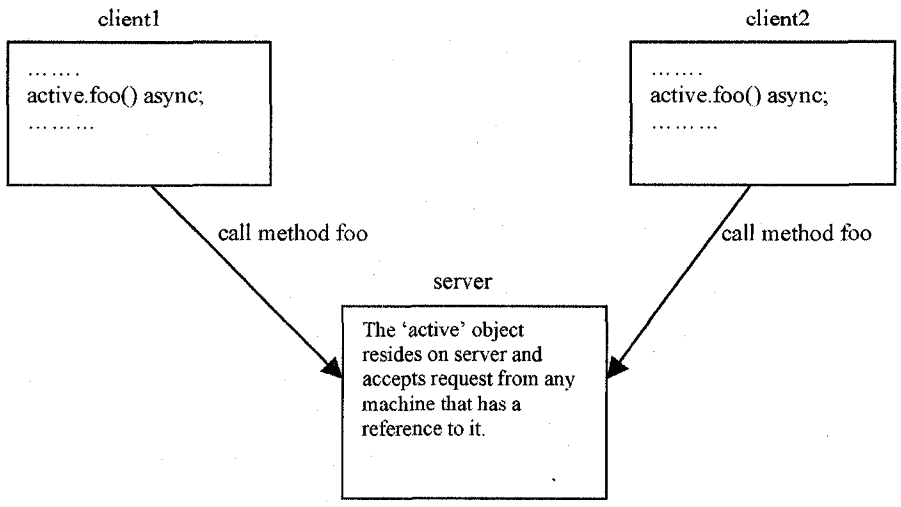

client1
…….
active.foo() async;
………
client2
…….
active.foo() async;
………
call method foo
call method foo
server
The 'active' object resides on server and accepts request from any machine that has a reference to it.

Figure 9: Invocations on the same object from different machines.

If **`client1`** and **`client2`** both have reference to the same object, **`active`**, that resides on the **`server`**, and invoke methods on the object no total ordering can be enforced. Even though **`client1`** may invoke a call on active object before **`client2`**, the requests can get to the server in a different order, due to the network traffic and the Java thread scheduling.

Even though no total ordering can be observed, partial ordering can. Restrictions apply only on invocations on the same active object from the same machine. If an invocation, **`foo`**, on an active object happens before another invocation, **`bar`**, on the same object on the same machine, then **`foo`** will be executed before **`bar`**. Of course **`bar`** does not have to execute immediately after **`foo`**, since other invocations from other machines on the same active object may get to it between **`foo`** and **`bar`**, but **`bar`** will definitely execute after **`foo`**.

We are simulating the asynchronous invocation by creating a thread that does the actual call in a synchronous manner and signals the main thread when the result is ready.

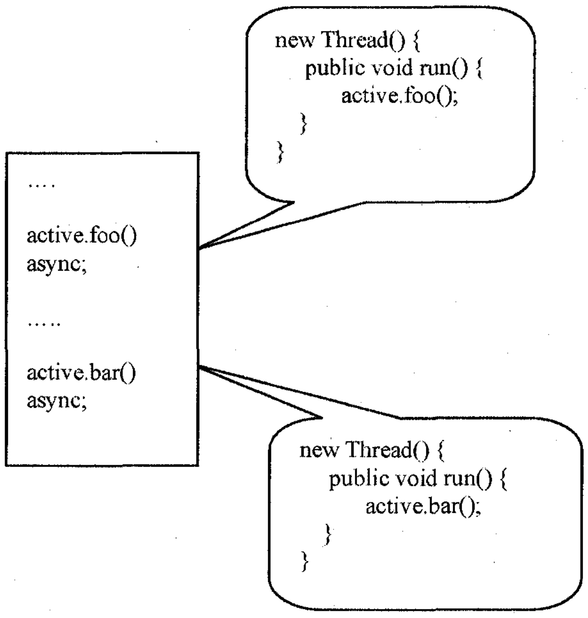


Figure 10: Simulating asynchronous calls.

The Java scheduling mechanism does not guarantee that the thread created first will be executed first. The use of priorities on thread would not assure the order of execution either. The Java scheduling mechanism pledges that every thread will be executed at some point, but no ordering is specified [15]. Under these circumstances we are using a message counter for each invocation. Recall that when a client calls an asynchronous method on an active object, it sends an invoke message to the server where the active object is located. This invoke message contains a message counter and the host name where the call was initiated from. This allows the server to execute the invocatinos in the appropriate order, even if the requests come out of order. Each active object has two queues:

- A queue of pending invocations, that keeps the invocations that are in the appropriate order and waiting to be executed
- A queue of out of order invocations, that stores the invocations are not in the appropriate order

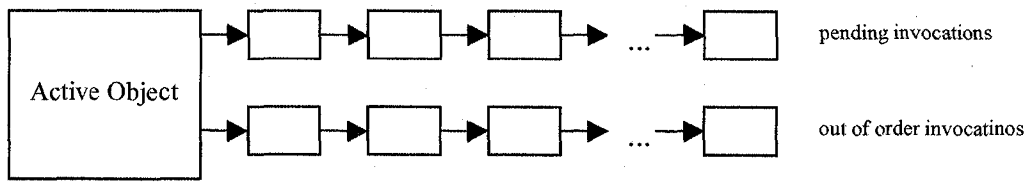


Figure 11: The invocation queues of active object.

Even if the server gets the invocations on the same object from the same machine in the wrong order, it assures that the partial ordering still holds. Something similar happens with the TCP packages that can get to the destination on different routes and thus can be out of order. The destination makes sure to put the packages in the correct order and reconstruct the initial message.

Active object keeps some additional information about each machine in order to be able to execute the messages, in according to the partial ordering. The list of computer/machine names that requested its services is stored internally by the active object and a corresponding message counter. From the active object's perspective this message counter identifies the message that has to execute for that host (to maintain the partial or local ordering). The active object receives an invoke message every time somebody requests its services. This message contains the method to be executed, the arguments, the name of the host that initiated the call and a message counter. The active object receives the message and looks for the host name in his list. If the host is in the list, it gets the message counter for that host and compares it with the message counter from the invoke message. If the two counters match, the request is added to the pending invocations queue and the message counter is incremented. If the message counter is greater than the expected counter then the message is put in the out of order queue, since other requests from that host have to be executed before this one.

The host may not be part of the active's object list of machines/computers/hosts. This happens the first time a host invokes active object's services. The host is added to the list of names and the message counter from the invoke message is compared with zero to determine if this is the message that has to be executed first. Based on this comparison, the message gets added either in the pending or out of order queue.

The active object executes the requests from the pending queue in order, on a first come first served basis. Once a message is executed, before proceeding to the next pending invocation, the active object checks the out of order queue. The following situation can occur: the active object receives two invocations from the same client. The expected message counter from this client is 19. The first invocation has the message counter 20. So, this request is placed in the out of order queue. The second invocation has the message counter 19 and it's placed in the pending queue and the expected counter is increased to 20. At some point the invocation with message counter 19 has to be executed, and once the execution is finished now the message with the counter 20 is the next one to be executed for this client so that invocation is moved from the out of order queue to the pending queue.

Client Manager

The core implementation of our active object runtime system is comprised by two classes: **`ClientManager`** and **`ServerManager`**. The communication between the machine where an asynchronous call is initiated and the actual machine where the active object resides is realized through these classes.

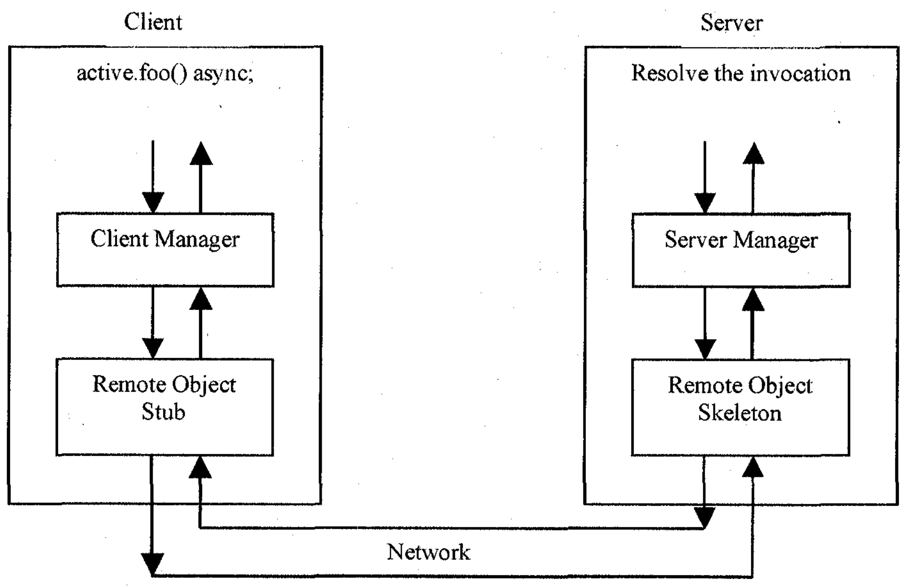


Figure 12: Communication between machines to fulfill the active object calls.

Figure 12 illustrates what happens behind the scenes when an asynchronous call is initiated. **`Client`** is the machine where the asynchronous method is invoked on object **`active`** and the **`server`** is where the object is located. The asynchronous call is translated at compile time to a ClientManager method invocation. How this translation happens and what is actually replaced with what will be detailed a later section of this chapter. The ClientManager identifies which machine it needs to communicate with and gets the remote object stub. It then sends the invocation to the remote stub which communicates with the remote skeleton (recall that RMI is used for interaction between machines) and finally the invocation gets to the ServerManager that resolves the call and sends back the result.

The core functionality of the ClientManager is implemented in two methods:

- **`invokeConstructor`** – which gets called when a new active object is created.
- **`invokeMehtod`** – is invoked every time an asynchronous invocation is executed on an active object.

Every time an active object is manipulated, either created or a method is invoked, the respective methods from ClientManager are invoked and the request is send to the active object (see Figure 12).

The ClientManager keeps track of all the active object invocations that are initiated from the machine that it is running on. As some of these invocations are executed asynchronously, the return values of these calls are also managed by the ClientManager. The ClientManager organizes the active object invocations based on the method where this calls were initiated. Recall that all methods are given a unique identifier that is used for organizing the active object invocations from within the method or when removing all the asynchronous calls that are not waited for. Inside a method more than one active invocation can exist. ClientManager organizes the active object invocations within a method based instance that is the target of the invocation. Figure 13 shows how ClientManager organizes the active object invocations:

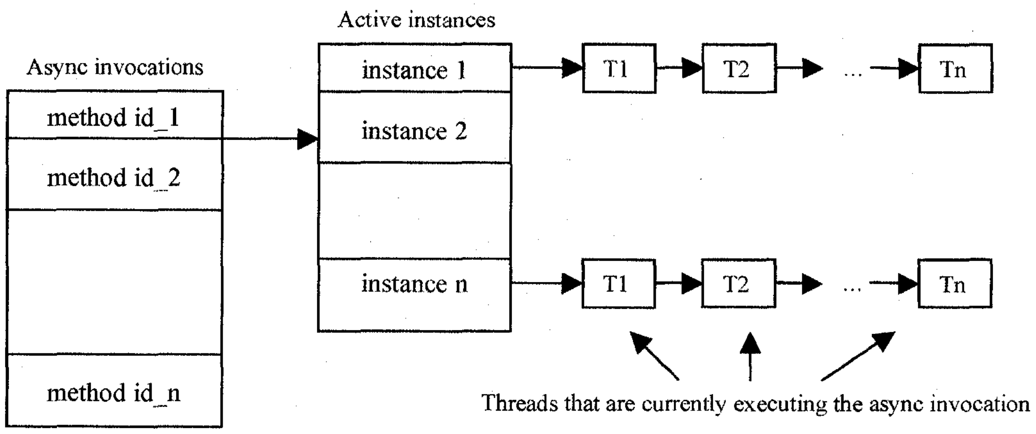


Figure 13: Maintaining asynchronous invocation on the client side.

The list of threads is composed of those threads that carry out asynchronous calls that return a value. Recall that whenever an asynchronous execution is invoked, a new thread is created and does the method call in a synchronous manner and signals the main thread when the result is available. While the thread executes the invocation, it remains in the list. Once the invocation finishes, the return value is saved and the thread is removed from the list.

Assume that the following code is executed and the method identifier for this method is five:

```
public void aMethod() {

    ActiveObj a = new ActiveObj() on "machine1";

    ActiveObj b = new ActiveObj() on "machine2";

    int val;

    a.foo() async;

    a.bar() async;
```

```
        b.foo() async;
        waitfor a val;
    }
```

Figure 14 shows what the client invocations will look like when the execution gets to the **waitfor** statement, assuming that none of the asynchronous invocations finish by that time:

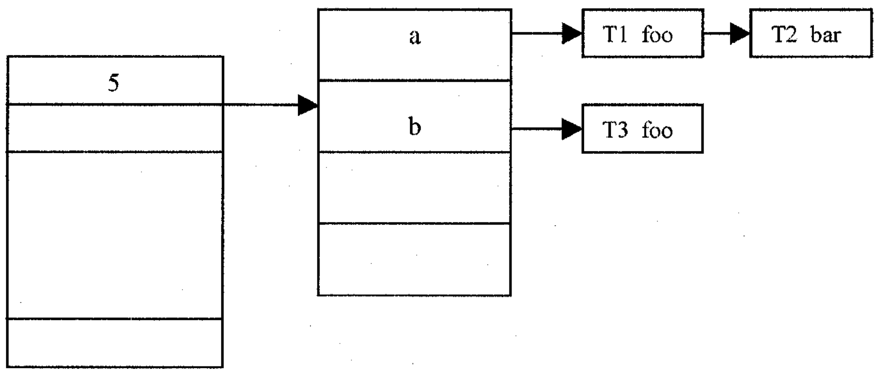


Figure 14: Information stored on ClientManager for the asynchronous invocations.

Besides the two core functions, **invokeConstructor** and **invokeMethod**, ClientManager also has some additional helper methods:

- **getMethodId()** – returns a unique identifier that is associated with every method that is executed on the machine.
- **removeUnwaitedCalls()** – takes a methodId as a parameter and removes all the asynchronous calls for which the programmer did not wait for the result. In the above example, once the method finishes, the manager gets all the instances that were used in the method body, *a* and *b*, and checks the list of threads. If a

thread is still in the list when the method finished, the value for that asynchronous call is discarded and the thread is moved to list of unwaited threads. The reason for moving the threads to another list is for the thread not to save the return value once it has finished. It may be the case, that when the method finished, the asynchronous call is still executing. In the above example, when **`aMethod`** finished, the asynchronous call, **`b.foo()`**, may still be executing. When the thread that carries out the invocation finishes, it first checks to see if itself is not in the list of unwaited calls, to determine if the result is saved or discarded. When **`aMethod`** finishes executing, both **`a.bar()`** and **`b.foo()`** will be dropped (we assume that both **`foo`** and **`bar`** return a value).

Server Manager

The ClientManager handles the invocations and initiates the communication for the client, where client is the machine that uses an active object. On the server side, the machine that holds the active object, the asynchronous or synchronous invocations are resolved by the ServerManager. The server managers accept **`create`** and **`invoke messages`** and send the requests to the corresponding active object that have to execute these requests. There is only one ServerManager on each machine involved in the execution process.

The ServerManager on each machine that participates in the execution of the program (will have at least one active object running on that machine) has to be started before the application is run. Starting the managers is done by calling a SecureShell (ssh) [12] script. This implies that all the machines have to support the ssh protocol. The reason why these managers have to be started is so they can make the remote objects that are used for communication available. Recall that all communication is done using RMI and the client has to first get the remote object for the server in order to send messages/invocations.

The ServerManager keeps a hash table of all the active objects that it manages (more than one object can be created on each machine). Each active object has a unique identifier that is used as a key to this hash table. The manager manipulates this hash table when it receives a message from the client. If it is a message to create a new active object, then the active object is created, associated with an identifier and added to the hash table. If the message is a request for a service (an invocation), the server looks up the active object that the request is addressed to and forwards it the message.

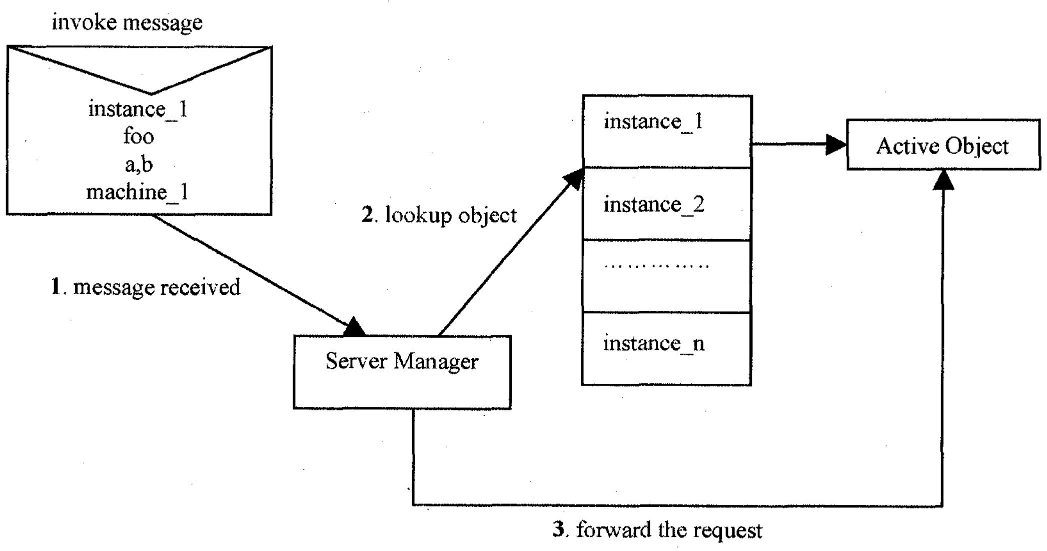


Figure 15: Invocation processing on the server side.

When the server receives the message from the client, the message contains the identifier of the object that was invoked, **`instance_1`** in the example. The server gets this identifier and looks it up in its table to obtain the actual instance that has to fulfill the request. In the above example the **`foo`** method needs to be invoked with the arguments **`a`** and **`b`** and the invocation comes from **`machine_1`**. The instance gets this request from the server and queues it in its list of invocations and at some point in time will execute it.

Compiler Modifications

All communication between the machine that makes the call on the active object and the machine that actually stores the instance that is the target of the invocation is made possible through compiler modifications. The ClientManager and the ServerManager are the core implementation of the runtime system. The compiler plays an important role as it accepts the new syntax and during desugaring phase (desugaring will be explain later on in this section) it replaces the new syntax with calls to this runtime system.

We have chosen to implement our active object system by adding new keywords to the Java language and modifying the Sun's open-source compiler to accept these new keywords and the new syntax. One other way to do it is by interface implementation or inheritance. For a class to be active it would have to implement a tag interface **`Active`**, just like Serializable interface allows objects of the class that implements it to be serialized and deserialized. This **`Active`** interface could have a **`waitfor`** method that would be called by active object when trying to get the result of an asynchronous method invocation. We chose to extend the language because it allowed us a much more natural approach. The modifiers describe the characteristics of a class, method or attributes. If the programmer wants to create a class that can be accessed by anybody, he simply adds the modifier public to the class declaration. When he wants to make a class active, he adds the **active** modifier to the class declaration. Also the creation of active objects and the asynchronous invocation is simpler and more natural with the extended language than it would have been if we would have used inheritance.

Chapter 3 explains the new keywords that were added and we will briefly present them again in this section:

- The **active** modifier can be used only in the class definition and marks all the instances of the classes that have this modifier in their definition as being active objects.
- New creation expression for active objects:

  ```
  new ActiveObj() on "machine";
  ```

  where **machine** is where the active object will reside.
- The **async** keyword can be used at the end of a method invocation on an active object to signal that the call will be asynchronous.
- A new statement that waits for the result of an asynchronous call:

  ```
  waitfor activeInstance variable;
  ```

The modified compiler performs two important tasks: first it checks if the syntax is correct and that the new keywords are used properly and in the allowed places and secondly, during the desugaring phase, it replaces the new syntax with calls to the ClientManager. The new keywords can be used only in some situations and the compiler guarantees that the syntax is correct. For example, the **active** keyword can only be used as part of the class definition and can not be part of an interface or method definition. All the methods from an active class can be called either synchronously or asynchronously so there is no need to use the active modifier in the declaration of a method. Also the methods that are part of a regular, passive class can not be invoked asynchronously. The compiler makes sure that the **active** keyword is used properly. The usage of the new creation statement, where the location of the object is specified, can only be used when instantiating active objects. Passive objects are created in the conventional way. The **async** keyword can only be used after a method invocation of an active object:

```
activeObj.foo() async;
```

and the result of an asynchronous call can be obtained by using the **waitfor** statement, that is newly added and has the following syntax:

```
waitfor activeObj variable;
```

The result of the **foo** invocation will be stored in the **variable**.

All these restrictions are verified by the compiler in the parsing and type checking phase and errors messages will be displayed if the syntax is not correct. The most important part of the compiling process happens in the desugaring phase. The developers of Java tried integrating new syntax/constructs to make programmer's life easier. An example of this type of new constructs is the **foreach** statement:

```
    foreach(String s : array) statement;
```

where **array** is an array of Strings. This simplifies the way an array is traversed. The programmer uses this construct, but during the desugaring phase this construct is replaced with a regular **for** syntax.

```
    for  (int i=0;i<array.length;i++) {
        String s = array[i];
        statement;
    }
```

The **foreach** construction only exists to simplify the iteration of an array and the compiler fills in the rest of the code that the programmer would normally have written. This happens during the desugaring phase.

The phases of the compiling process in the order they happen are: parsing, attribution, type checking, desugaring and generation. At the moment of desugaring, the compiler guarantees that the program is syntactically and semantically correct. Following the model of the **foreach** construct that is replaced with some other code during desugaring, we are replacing the new keywords and new syntax with regular Java code (actually calls to our ClientManager).

Modifying the Active Keyword

The **active** keyword, which is part of the class declaration, is removed during the desugaring phase. During the attribution phase additional information is saved in the node that represents a class, to identify that the class is active. The extra piece of information that is stored in the node that represents the class is the **machineName**. If the **machineName** is null, then the class represented by that node is a regular passive object. Otherwise, it is an active class. When we create an active object, we can still use the regular syntax:

```
    activeObj = new ActiveClass();
```

In this case the active object is created on the current machine and the **machineName** will have the value different than null, namely localhost.

Modifying the New Creation Expression

Active objects can be created either using the new construct where the machine that will hold that instance is given or the regular, conventional way:

```
ActiveClass obj = new ActiveClass() on "machine_1;  (1)
ActiveClassj obj = new ActiveClass();               (2)
```

If the regular syntax (2) is used, then during the type checking phase the compiler sees that an active object instance is created and the machine name is not specified. If no machine name is specified then localhost is used as the default location. So the regular syntax from above, when creating active objects, is identical to:

```
activeObj = new ActiveClass() on "localhost";
```

Then during the desugaring phase the new creational syntax is replaced by a static call to the ClientManager. The call above that uses the uses the new syntax (1), will be replaced with:

```
InstanceInfo obj = ClientManager.invokeConstructor(
                                  "ActiveClass",
                                   new Object[]{},
                                   machine_1");
```

The **`invokeConstructor`** method of ClientManager receives the type of the class that has to be created, **`ActiveClass`**, the arguments for the constructor, in this case none, and the machine name where it has to create the instance, **`machine_1`**. Then the ClientManager generates a **`create message`** and sends it to the ServerManager of **`machine_1`**. The invoke constructor method returns an object of type **`InstanceInfo`** and it is an object created by **`machine_1`**. This object represents a 'remote reference' to the active object, meaning it contains information necessary to access the object on **machine_1**.

Adding the MethodId

Recall that when invoking an asynchronous method of an active object, a thread is created and executes the call synchronously while the main thread continues the execution. These threads/invocations are organized on the client side based on the method where the call was initiated. For this to be possible, each method has a unique method identifier. Each method that has in its body an asynchronous invocation will have a special method inserted as the first statement:

```
    Long methodId = ClientManager.getMethodId();
```

This statement will be inserted as the first statement of the method body. It will be inserted as the second statement in the constructors, since **this()** or **super()** have to be the first statement from the method body.

```
    public  class ActiveObj extends OtherObj {
        public ActiveObj() {
        //here some code that invoke an active object
       }
    }
```

After the desugaring class the code will look this:

```
    public  class ActiveObj extends OtherObj {
      public ActiveObj() {
        super(); //added automatically by the compiler
        //added by our compiler
        Long methodId = ClientManager.getMethodId();
        //here some code that invoke an active object
      }
  }
```

The methods that do not have any active object invocations inside their body will not have the variable declaration of the **methodId** inserted. Determining if a method has an active object invocation is done prior to desugaring in the attribution phase.

Modifying the Waitfor Statement

The **waitfor** statement is used to suspend the main thread to obtain the result of a previous asynchronous invocation. The syntax of the **waitfor** statement is:

```
waitfor activeInstance variable;
```

where the **activeInstance** represents the instance of the asynchronous invocation, and the **variable** will hold the return result of that invocation. The initial format of the **waitfor** statement was:

```
variable = waitfor(activeInstance);
```

The problem with this format is that during the type checking phase, the type of the **waitfor** method could not be determined. More than one method invocation can be called on the **activeInstance** and the type of the **waitfor** should be the return type of the method invoked on the instance but that cannot be determine at the compile time.

That is the reason why we chose the first **waitfor** format. The **waitfor** statement will also be replaced at desugaring time with the following block:

```
{
ReturnObject returnObject0=ClientManager.waitForThread
```

```
                                              (methodId,
                                              activeInstance);
    variable = (Integer)returnObject0.getReturnValue());
    }
```

The first statement from this block waits for the thread that carries out the execution to complete and returns a **ReturnObject**. The **ReturnObject** will contain the actual return value of the invocation and the result can be obtained by calling the **getReturnValue** on the **returnObject**. A cast is also done to the type of the **variable**. In the above case the return value of the asynchronous invocation that is waited for is of type Integer and the type of the **variable** is also Integer (or int because autoboxing is used).

Modifying the Invocations on Active Object

The invocations of the active object can be executed synchronously or asynchronously, depending if the **async** keyword is present at the end of the method call:

```
val = activeObj.foo(b);//sync. foo returns a value
activeObj.bar(a) async; //async. bar returns a value
```

The syntax for the synchronous and asynchronous invocation is somehow different if the method invoked returns a value. The synchronous call has to assign the return value to a variable, while the asynchronous call does not. The return value of an asynchronous call will be assigned using the **waitfor** syntax. The compiler enforces these rules and will complain if they are not respected.

We will first discuss how the asynchronous calls get translated during the desugaring phase. The **bar** call from the above example, will be replaced by:

```
ClientManager.invokeMethod(methodId, activeObj, "bar",
                                     new Object[]{a},true);
```

The new code calls the **invokeMethod** of ClientManager and passes the following parameters:

- The **methodId** - used for managing invocations from this method body.
- The 'remote reference' to the active object, **activeObj**.
- The method to be executed, **bar**.
- The arguments passed to the method, **new Object[] {a}**.
- A boolean value that specifies if the method has a return value or not. In our example, the **bar** method returns a value.

The synchronous call, `val = activeObj.foo(b)`, gets translated in the following block:

```
{
ClientManager.invokeMethod(methodId, activeObj, "foo",
                                     new Object[]{b}, true);
ReturnObject returnObject=ClientManager.waitForThread(
                                          methodId, activeObj);
val = (Integer)returnObject.getReturnValue();
}
```

The synchronous call is actually composed of the following: an asynchronous call followed immediately by a **waitfor** statement that gets the result. The first line of the translated block represents the asynchronous call and the following two lines correspond to the **waitfor**. The **async** keyword allows the programmer to specify when the waiting for the result is done and he gets the result using the **waitfor** statement. If the **async** keyword is missing, and the method returns a value, then the result is waited for immediately after the call. If the method called does not any value, then the whole block is replaced by the first line, since no waiting for the result is necessary.

Removing the Unwaited Asynchronous Calls

It may happen that the programmer executes an asynchronous call and forgets to wait for the result of that call. When a method finishes its execution, all the results of the unwaited calls will be ignored and discarded. This is done by adding at the end of the method body or before each return statement a call to discard all the asynchronous calls that were initiated from this method and not waited for. This call is:

```
    ClientManager.removeUnwaitedCalls(methodId);
```

This statement is only added, just like the statement creating the **methodId**, only for those methods that have at least a call on an active object.

An Example

The class TestActive creates some active objects of type **ActiveClass** and makes asynchronous and synchronous calls to those active objects.

```
//the definition of the ActiveClass
public active class ActiveClass {

  public void foo(){
    System.out.println("foo was called ");
  }

  public Integer bar(Integer s){
    System.out.println("bar was called "+s);
    return s*2;
  }
}
```

Figure 16: The definition of an active class.

```
//a class that shows how the new keywords are translated into regular Java syntax
public class TestActive {
  public TestActive() {
    //create an active object on the local machine
    ActiveClass a = new ActiveClass();
    //invoke the bar method asynchronously
    a.bar(new Integer(4)) async;
    System.out.println("computing");
    Integer val=null;
    // saving the return value of bar in the val variable
    waitfor a val;
  }

  //this method does not have any active object invocations, so no modifications are done  to the method
  public void noCallOnActive(){
    System.out.println("no call on active");
  }

  //this method has active object invocations, so the method body will be modified
  public boolean callOnActive() {
    //create an active object on the local machine
    ActiveClass a = new ActiveClass();
    //invoking the bar method asynchronously
    a.bar(new Integer(4)) async;
    System.out.println("computing");
    //invoking the foo method synchronously
    a.foo();
    Integer val=null;
    //waitfor the result of the async invocation of bar
    waitfor a val;
    if (val == 3 ) {
      return true;
    } else {
      return false;
```

Figure 17: The definition of a class that uses an active object.

The **TestActive** class has a constructor and inside the constructor body an active object invocation is executed. The compiler adds by default the **super()** call and this called is followed by a declaration of the **methodId**. In **callOnActive** this declaration is the first statement from the method body. If no machine name is provided when creating an instance of an active class, the active object is created on the local machine. This is how these classes will look after the code has been compiled using our modified compiler:

```
//the active keyword was removed from the class definition
class ActiveClass {
   ActiveClass() {
      super(); //added by the compiler
   }
   public void foo() {
      System.out.println("foo was called ");
   }
   public Integer bar(Integer s) {
      System.out.println("bar was called " + s);
      return Integer.valueOf(s.intValue() * 2);
   }
```

Figure 18: The new definition of ActiveClass generated by our compiler.

```
public class TestActive {
  public TestActive() {
    super();//added by default by the compiler
    // added by the compiler due to active invocations in the method body
    Long methodId = ClientManager.getMethodId();

    //the translation of the new creation expression
    InstanceInfo a = ClientManager.invokeConstructor("ActiveClass", new Object[]{}, "localhost");

    //the translation of an asynchronous invocation
    ClientManager.invokeMethod(methodId, a, "bar", new Object[]{new Integer(4)}, true);
    System.out.println("computing");
    Integer val = null;
    //the translation of the waitfor statement
    {
      ReturnObject returnObject0 = ClientManager.waitForThread(methodId, a);
      val = (Integer)returnObject0.getReturnValue();
    }
    //added by the compiler to remove all the unwaited calls that originated in this method
    ClientManager.removeUnwaitedCalls(methodId);
  }

  //no modifications on this method, since no active object invocations are executed
  public void noCallOnActive() { System.out.println("no call on active"); }

  public boolean callOnActive() {
    Long methodId = ClientManager.getMethodId(); //added by the compiler

    //the translation of the create expression
    InstanceInfo a = ClientManager.invokeConstructor("ActiveClassj", new Object[]{}, "localhost");

    //translation of the async invocation
    ClientManager.invokeMethod(methodId, a, "bar", new Object[]{new Integer(4)}, true);
    System.out.println("computing");

    //translation of the synchronous invocation that returns void
    ClientManager.invokeMethod(methodId, a, "foo", new Object[]{}, false);
    Integer val = null;
    //the translation of the waitfor statement
    {
      ReturnObject returnObject1 = ClientManager.waitForThread(methodId, a);
      val = (Integer)returnObject1.getReturnValue();
    }
    if (val.intValue() == 3) {{
        ClientManager.removeUnwaitedCalls(methodId); //add this statement before each return stmt
        return true;
      }} else { {
          ClientManager.removeUnwaitedCalls(methodId);
          return false;
        }}
  }
}
```

Figure 19: The new definition of TestActive generated by our compiler.

Problems with the Implementation

When a method is invoked asynchronously (adding the **`async`** keyword at the end of the method invocation), the class of the object has to contain the **`active`** keyword in its declaration. Only the actual type of the object is checked to be active and no checking is done on its supertypes. For example, if a class *Shape* is declared to be active and this class is extended by another class, *Square*, that is not declared to be active, when an asynchronous invocation is made on a *Square* instance, an error message will be thrown by the compiler. During the type checking we only check if the *Square* type is active and we don not verify its class hierarchy. The active property of a class does not propagate down in the class hierarchy (unlike serializable property).

The methods from an active class can throw any type of checked exception. Type exceptions are either caught or explicitly thrown (declared in the declaration of the method). If an active object method throws an exception, then all invocations of that method, either synchronous or asynchronous, must be surrounded by a try catch block or be declared in the method declaration. This is enforced by the compiler. The synchronous and asynchronous invocations are replaced by a call to the ClientManager's **`invokeMethod`**. This method catches all the exceptions that are thrown by the active object invocation, but do not forward them to the thread where the invocation originated. Instead it prints an error message and exits the program.

CHAPTER 5

ACTIVE OBJECTS FOR DISTRIBUTED COMPUTING

A distributed computing system is composed of multiple software components that run on different computers connected by a network, but these components work together as a single system. The computers can be physically close together, located in the same room or building, or geographically distant and connected by a wide area network. Tasks or jobs can be distributed on different machines where they can be executed in parallel, thus increasing the number of tasks executed in a unit of time. A bank can be viewed as an example of a distributed system, where the bank tellers are the processors and the customers are the tasks the processors have to execute [17]. If a bank has only one teller (single processor), then the tasks have to be executed serially one after another (only one customer can be helped at any point). When more than one teller is available, the bank may be operated as a distributed system; tasks (customers) can be executed by the next available teller and a teller can execute its task in parallel with other tellers. A parallel system is a tightly coupled distributed system where all the resources are utilized to solve a single problem [17]. The distributed systems are more general and may simultaneously work on many problems.

Some may think that if a problem takes ten seconds to be solved on a single processor, then using a parallel system with ten processors it should take only one second. This is not always the case. A parallel system has some communication time that has to be taken into consideration, beside the processing time. Also not all parts of the program can be parallelized or parts of the problem have to be executed sequential (more on this in next chapter). The communication time is the time it takes for a message to get from one processor to another. This communication time depends on the amount of data that has to be transmitted (how big the message is) and the network bandwidth and latency. The ratio of processing time (the time it takes a processor to finish the task) to communication time is good to be as big as possible (that means more time is spent computing than communicating). This ratio also influences the speedup (the ratio of the execution time using *1* processor and the execution time using *n* processors). We will present more on speedup in the following chapter.

Distributed and parallel systems can be divided into two categories based on processors intercommunication. A *shared-memory* system where a global memory is shared by all the processors and the communication between processors is done by writing and reading this global memory. The second category is a *message passing* system where the communication between processors happens by explicitly exchanging messages. In the actor system, which was used as a model for our active object system, the communication between actors is done by sending and receiving messages.

The implementation of our active object system allows parallel computation, that is active objects can executed their methods in parallel. More over, the active objects do not have to be located on the same machine, but can be distributed on different machines. The distributed support offered by Java is the Remote Method Invocation. When the programmer wants to use RMI, the following steps must be taken [10]:

- Declare an interface with all the methods that can be accessed remotely. This interface has to extend the *Remote* interface.
- Provide implementation for all the remote methods.
- Create and install a secure manager.
- Register at least one remote object with the RMI registry (or other naming services) for bootstrapping purposes.
- The client has to lookup the remote object before invoking methods on it.

Our system reduces the work that a programmer has to do in order to use RMI. All the programmer has to do is to create an active class (that would be the remote class) and invoke methods on object of that class synchronously (the RMI calls are synchronous). The active object's methods can be invoked asynchronously as well, but a typical RMI call is synchronous: the programmer gets the remote object, call a method on it and wait for the result. This is the exact behavior obtained when an active object's method is called synchronously. The programmer can specify the remote machine that will execute all the invocations with the use of the new creation expression:

```
remoteObj = new RemoteClass() on "machine";
```

The programmer can create another instance of **RemoteClass** on a different machine by specifying another machine name after the **on** keyword. Some argue the programmer has to interact differently with the distributed objects compared to the non-distributed objects [16]. When using on object that is located on a remote machine the programmer has to be aware of latency and failure problems that may arise. In our system the method invocations of an active (distributed) object is done in the same way as the invocations on a regular, passive object. Our system uses RMI for the communication between objects situated on different machines, so a *RemoteException* is thrown any time a failure occurs. Methods on an active object can be invoked asynchronously (by adding the **async** keyword at the end of the method invocation). The caller's thread does not wait for the invocation to finish and continues executing the rest of the code, while the method invocation executes in parallel. The overhead associated with the distribution or parallelization of the application or the task is reduced by using the **async** keyword, because the client can continue with some other processing during the time the asynchronous call is executed.

Master/Slave Model. The Mandelbrot Set Example.

One often used pattern is distributed systems is the master/slave architecture. In this model, the master divides the work into tasks and sends the tasks to the slave(s). The slaves execute the tasks and send the results back to the master. Our active object system extends the master/slave architecture offered by the OO model by allowing the master to send messages to the slaves asynchronously. The master can continue executing other tasks while the slaves are executing their tasks. Also, a slave can execute a task in parallel with other slaves or with the master. When an object **M** (master), creates an object **S** (slave), using the new creation expression, **M** can send **S** the task it has to execute. **M** does not have to block until **S** finishes; it continues executing the rest of its code.

We will now show how our active object system can be used to compute the Mandelbrot set utilizing a master/slave architecture. The Mandelbrot set is made up of complex points **c**, for which the complex quadratic polynomial: $x_{n+1} = x_n^2 + c$ (1) remains bounded. The following assumption is made: $x_0 = 0$ for every point **c**. For

Mandelbrot this condition can be translated to: after applying formula (1) the absolute value (modulus) remains bounded (in our case bounded means less than 2). If we consider the following complex point, $x = 1$, then the first 4 iteration of formula (1) will look like this:

$x_1 = x_0 + 1 = 0 + 1 = 1;$

$x_2 = x_1 + 1 = 1 + 1 = 2;$

$x_3 = x_2 + 1 = 4 + 1 = 5;$

$x_4 = x_3 + 1 = 25 + 1 = 26;$

After each iteration the value of x_n grows, so it is not bounded. That means that the complex point $x = 1$, does not belong to the Mandelbrot set. If compute the same four iterations for the complex number, $x = i$, we get the following results:

$x_1 = x_0 + i = 0 + i = i;$

$x_2 = x_1 + i = -1 + i = i-1;$

$x_3 = x_2 + i = -1 - 2i + 1 + i = -i;$

$x_4 = x_3 + i = -1 + i = i-1;$

The results show that no matter how big n is (where n represents the number of iterations), the value of x_n remains bounded and less than two. So the complex value $x = i$ belongs to the Mandelbrot set.

The formula (1) also shows that determining whether a point is in the Mandelbrot set or not, is independent of the adjacent points. In order to parallelize the computation of the Mandelbrot set, each processor will receive a portion of the complex plane and perform the necessary computation for the points in the assigned section.

```
//this class does the actual Mandelbrot computation.
//It is an active class, because the active modifier is part of the class declaration
public active class Mandelbrot {
  //the maximum number of interations when computing the value for a complex numer
  private static final int MAX = 256;

  public MandelbrotSet compute(Integer startLine, Integer noOfLines, Integer width, Integer
                height, Float real_min, Float real_max, Float imag_min, Float imag_max) {

    Integer[][] result = new Integer[width][noOfLines];
    ComplexNo c;

    //compute the scale factor
    float scale_real = (r_max - r_min) / width;
    float scale_imag = (i_max - i_min) / height;

    //go through all the points, and compute the mandelbrot
    for (int y = startLine; y < noOfLines + startLine y++) {
      for (int x = 0; x < width; x++) {
        //create the complex number
        c = new ComplexNo(real_min + ((float)x * scale_real), imag_min + ((float)y * scale_imag));
        //compute the value for the complex number
        Integer val = cal_pixel(c);
        //save the value in the result matrix
        result[x][y - startLine.intValue()] = val;
      }
    }
    return new MandelbrotSet(result);
  }

  //calculate the value of the complex number
  private Integer cal_pixel(ComplexNo c) {
    int i = 0;
    float tmp, lengthsq = 0;
    ComplexNo z = new ComplexNo(0, 0);
    do {
      tmp = z.real * z.real - z.imag * z.imag + c.real;
      z.imag = 2 * z.real * z.imag + c.imag;
      z.real = tmp;
      lengthsq = z.real * z.real + z.imag * z.imag;
    }     while ((++i < MAX) && (lengthsq < 4.0));
    return new Integer(i);
  }
}
```

Figure 20: The implementation of the Mandelbrot algorithm.

Figure 20 shows the code for the class that implements the Mandelbrot algorithm. The **compute** method receives a portion of the complex plane; it checks the complex quadratic polynomial value for every point in this plane. If the value is bounded (less than two) than it belongs to the Mandelbrot set and the point is saved in the result. The **cal_pixel** method computes the quadratic polynomial value of a point from the complex plane.

```
//active objects pass it as a result of the compute method
//it is passed across the network so it has to be serializable
public class MandelbrotSet implements Serializable {

  private Integer[][] result;

  public MandelbrotSet(Integer[][] result) {
     this.result = result;
  }

  public Integer[][] getResult() {
     return result;
  }
}
```

Figure 21: The class that stores the Mandelbrot set.

```
//the class represents a complex number
public class ComplexNo {
  public float real;
  public float imag;

  public ComplexNo(float real, float imag) {
     this.real = real;
     this.imag = imag;
  }
}
```

Figure 22: A Class representing a complex number.

The classes from Figures 21 and 22 are helper classes. The **MandelbrotSet** saves the result of the computation and it is used by the slave to send the result to the master. The **ComplexNo** class represents a complex number and is passed as a parameter to the **cal_pixel** method.

```
public class MandelbrotClient {
   private static int computerNo;
   private static String[] computerNames;
   public int[][] map;
   public int disp_width, disp_height;
   public float real_min, real_max, imag_min, imag_max;
   public String inputFile, outputFile;

   public static void main(String[] args) throws Exception {
      MandelbrotClient mandel = new MandelbrotClient();
      mandel.init(args);

     //nr of lines computed by each process
      int nrofLines = mandel.disp_height / computerNo;

      Mandelbrot[] mandelbrot = new Mandelbrot[computerNo];
      for (int i = 0; i < computerNo; i++) {
         //create the active objects and send them the workload
         mandelbrot[i] = new Mandelbrot() on computerNames[i];
         mandelbrot[i].compute(startLine + i*nrofLines, nrofLines,
                               mandel.disp_width, mandel.disp_height,
                               mandel.real_min, mandel.real_max,
                               mandel.imag_min, mandel.imag_max)   async;
     }

     //get the results for all the active objects and write them to the output file
     MandelbrotSet mandelSet = null;
     for (int i = 0; i < computerNo; i++) {
        waitfor mandelbrot[i] mandelSet;
        Integer[][] result = mandelSet.getResult();
        //write the result to file
     }

     private void init(String args[]) {
       //initialize disp_width, disp_height, real_min, real_max, imag_min, imag_max, inputFile, outputFile
     }
}
```

Figure 23: The master's code that computes the Mandelbrot Set.

Figure 23 shows the code the master executes. The master computes the Mandelbrot set of a complex plane, by dividing the complex plane into subplanes and sending the subplanes to the slaves to do the actual computation. After reading all the input data (the left top corner and the right bottom corner of the complex plane, the slaves, and the input and output file) the master creates the slaves. The first `for` loop creates the slaves and sends each slave the portion of the complex plane that it has to compute. The slaves are computing in parallel, since the quadratic value of each point in the complex plane can be calculated independently of any other points in the plane. The slaves do not have to pass any information to each other to perform the computation. The only communication is between the master and the slaves. The master has to get the results from the slaves and this is done in the second **`for`** loop.

As stated previously, the master computes only a portion of the complex plane. The top left corner and the bottom right corner are passed as an argument to the main function as well as the name of the machines (slaves) that will participate in the computation. The execution time of this program (with the number of slaves and the dimensions of the complex plane varying) is presented in the next chapter. Also these execution times are compared with the execution time of the sequential algorithm.

This Mandelbrot example shows the way that our system can be used to implement distributed and parallel applications. The master/slave distributed model can be implemented with active objects. The slave is the active object (must have the **`active`** keyword in the class definition) and the master is the object that creates the active object. The master sends tasks to the slave by invoking one of the slave's methods either synchronously or asynchronously (using the **`async`** keyword). The slave can in turn become a master for other objects (namely for those active objects that the slave creates).

Parallel applications can be developed in Java using threads. The overuse of *synchronized* (to protect the access to a shared region) keyword can some time lead to deadlocks that are hard to find. Our active object model is an alternative to developing parallel and distributed (since active objects can be created on different computers on the network) applications in Java. The next chapter will evaluate the performance of applications developed using active objects.

CHAPTER 6

RESULTS

In order to test our active object system and to measure its efficiency (by comparing it to the sequential version run time) we have implemented three programs: Mandelbrot Set computation, Matrix Multiplication and Pipeline processing. The results obtained are presented in the following subsections. The sequential versions were implemented first and run ten times. Out of these run times the best was selected. The parallel versions were implemented using active objects. When testing the parallel versions, the number of processors that participated in the execution was varied. For each variation, the program was run ten times and the best run time was selected, just like the sequential version. The sequential version used was not the parallel version ran only on one processor, but a separate, single threaded implementation. For both the sequential and the parallel versions, the size of the problem was also varied and tested ten times, out of which the best run time was selected. The execution times presented in the following tables are expressed in seconds.

Potential for Increased Speedup

When evaluating the performance of a parallel application another metric is used beside execution time: speedup. This metric shows how much faster the parallel program is compared to the best solution (sequential algorithm) on one processor.

$$S(p) = \frac{Ts}{Tp} = \frac{time\ of\ best\ sequential\ program}{time\ of\ parallel\ program}$$

The maximum speedup for p processors is usually p (linear speedup). This is achieved when the computation divides equally into p parts without overhead. Superlinear speedup is sometimes obtained due to the cache effect from different memory hierarchies. If the processors have a big cache memory, then more or even all core data can fit into the cache reducing the memory access time.

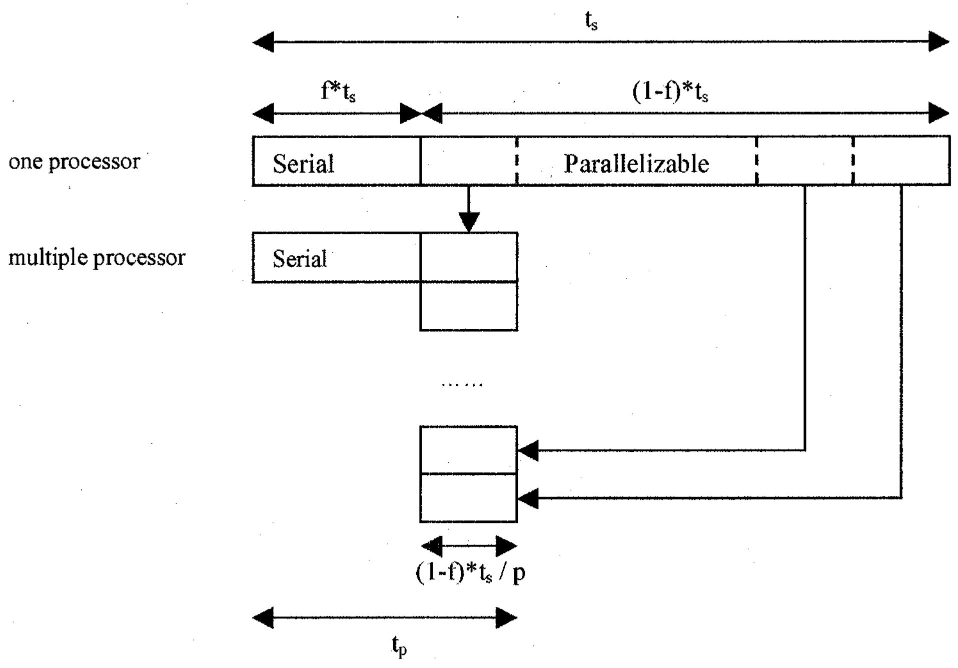


Figure 24: Dividing the parallelizable part of a program to p processors.

The linear speedup is not always possible to achieve due to several factors: periods of time exist when not all processors can perform computation, extra computation in the parallel version, communication time between processors. Another factor is that a fraction f of the computation can not be divided into concurrent tasks as presented in Figure 24.

Thus the speedup factor can be expressed by the following formula, known as Amdahl's law:

$$S(p) = \frac{p}{1+(p-1)*f}$$

When dividing the program on infinite number of processors (p tends to infinity) then the speedup tends to $\frac{1}{f}$. For example, if $f = \frac{1}{20}$, then the maximum speedup is 20.

Mandelbrot Set

The Mandelbrot set computation was described in the previous chapter. More over, the implementation of Mandelbrot using our active objects system was also presented and explained (see APPENDIX A for details of the implementation). Now we present and compare the results of the sequential version of Mandelbrot versus the parallel and distributed version. Recall that the Mandelbrot set verified all the points in a complex plane to remain bounded when calculating the quadratic polynomial value. The bigger the complex plane that has to be verified is, the more complex points have to be analyzed, which results in longer execution time. The size of the complex plane is considered to be the size of the problem. In order to get a better evaluation of the performance of the parallel version, we vary the size of the problem. We start with a complex plane of 5,000 x 5,000 points and increased each dimension of the complex plane by 5,000 points each time, up to 20,000 x 20,000 points. Additionally, the parallel versions were tested on 4, 8, 16, 32 and 64 slaves (plus a master) for problem size.

Table 1 Execution time of the sequential and parallel versions of Mandelbrot Set

	5000 x 5000	**10000 x 10000**	**15000 x 150000**	**20000 x 20000**
Sequential	18.8	74.9	169.3	300
1 master+4 slaves	6.6	22.5	57.2	105
1 master+8 slaves	4.5	14.1	31.4	56
1 master+16 slaves	3.9	10.4	21.5	36.7
1 master+32 slaves	4.8	12.1	19.9	30.8
1 master+64 slaves	7.5	13.6	21.7	32.8

Table 1 shows the results obtained after running the sequential and the parallel versions and represent the best run time out of ten tries. The results demonstrate that the parallel versions perform better all the time, no matter how many processors (slaves) are used in the computation, but adding more processors does not mean that the speedup will grow as well (see Figure 25). Another important thing that can be noticed is that by increasing the processor numbers, the execution time decreases but only up to a point. For a complex plane of 5,000 x 5,000, when 16 processors are used, a shorted execution time is obtain than when using 32 processors. The number of processors that need to be used to solve the problem in shorter amount of time varies with the size of the problem; it can be seen that for a problem size of 20,000 x 20,000, 32 processors perform better than 4 or 16 (5,000 x 5,000 gets the best execution time when 16 processors are used).

It is also important to mention that no input/ouput operations were taken into consideration when calculating the run time of the Mandelbrot program (the I/O is that fraction of the code that can not be executed in parallel).

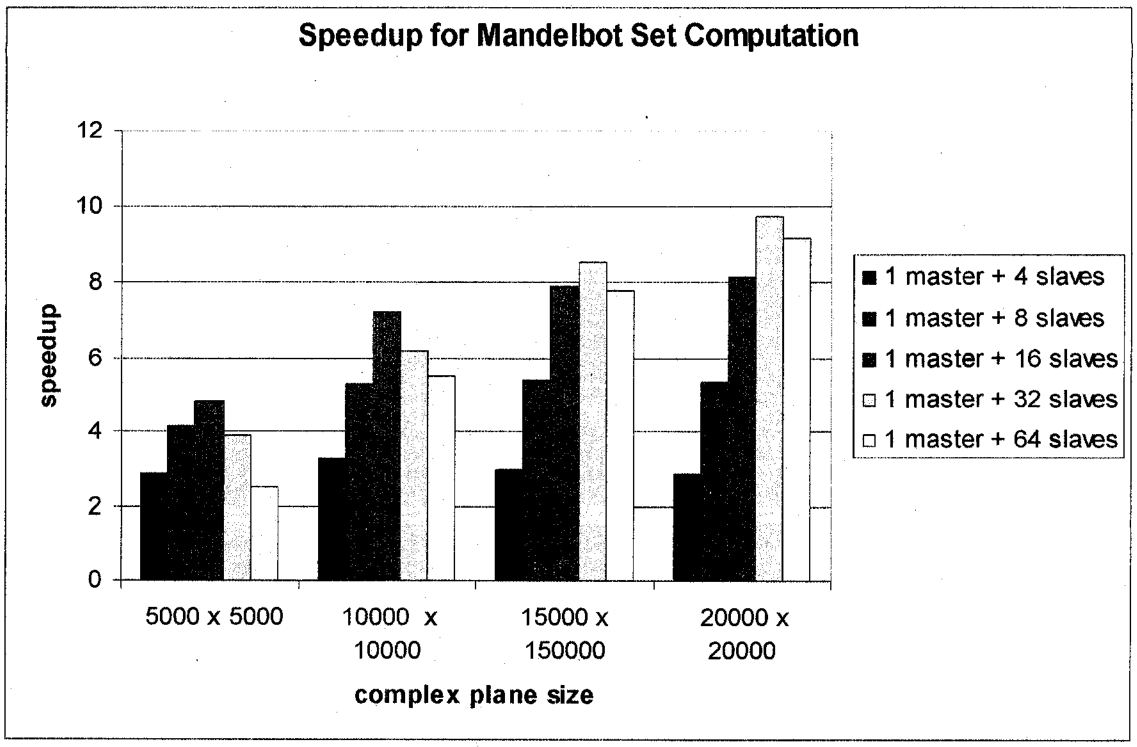


Figure 25: Speedup for Mandelbrot Set.

Figure 25 shows that no linear speedup was obtained for our Mandelbrot parallel version. One reason why the linear speedup was not obtained is because of the overhead of the communication between the master and the slave. There is a small overhead when a master creates a slave, and then it is the overhead of transmitting the result back to the master. Even if no linear speedup was obtained, the parallel version performs better than the sequential version no matter of the problem size.

Matrix Multiplication

Matrix multiplication is another problem that can be implemented in parallel. The matrices are divided in submatrices (similar to a grid) and each processor gets one submatrix A and one submatrix B. An example of how a matrix is divided in submatrices is shown in Figure 26.

A1	A2	A3	A4
A5	A6	A7	A8
A9	A10	A11	A12
A13	A14	A15	A16

Processor 1 gets A1
Processor 2 gets A2
Processor 3 gets A3
……………………
Processor 16 gets A16

Figure 26: The division of a matrix in submatrices.

Before the parallel computation is started, each processor gets its corresponding submatrix A and submatrix B. The following steps are executed (see APPENDIX B for details of the implementation):

- Each processor sends its A submatrix to the neighbor process from the right (with wrapping around, A4 sends to A1). This process is called piping. The source of piping is different at every step.
- Each processor computes matrix multiplication of its A and B submatrices.
- Each processor sends its B submatrix to the neighbor processor above (with wrapping around, B1 sends to B13). This process is called rolling. The source of the rolling is different at every step.

The results obtained after running the sequential and parallel versions of the matrix multiplication are shown in Table 2.

Table 2 Execution time of the matrix multiplication

	512 x 512	1024 x 1024	2048 x 2048	4096 x 4096
Sequential	2.3	28.6	283.3	6983.2
1 master+4 slaves	3.6	21.1	176.3	1728.8
1 master+16 slaves	4.5	20.4	144	1355.8
1 master+64 slaves	9.6	26.2	150.9	1305.1

The results demonstrate that the parallel version does not always perform better than the sequential one (the parallel version of Mandelbrot always performed better than the sequential version). The reason why this happens is because of the overhead of starting each processor and of passing submatrices A and B around. Also the synchronization between processors will cause a slowdown for small datasets because more time is spent communicating then computing. Submatrix A2 has to finish its computation (from step 2) before being able to receive the submatrix of A1, copy it and forward it to A3. Even if A3 has finished its computation faster than A2, it can not proceed until A2 is executing the piping process. As soon as the problem size is increased (the size of the matrix), the parallelism starts paying off: for 2,048 x 2,048 the parallel version with 16 processors executes the program in half of the sequential execution time (obtaining a speedup of 2). Figure 27 shows the speedups for the matrix multiplication.

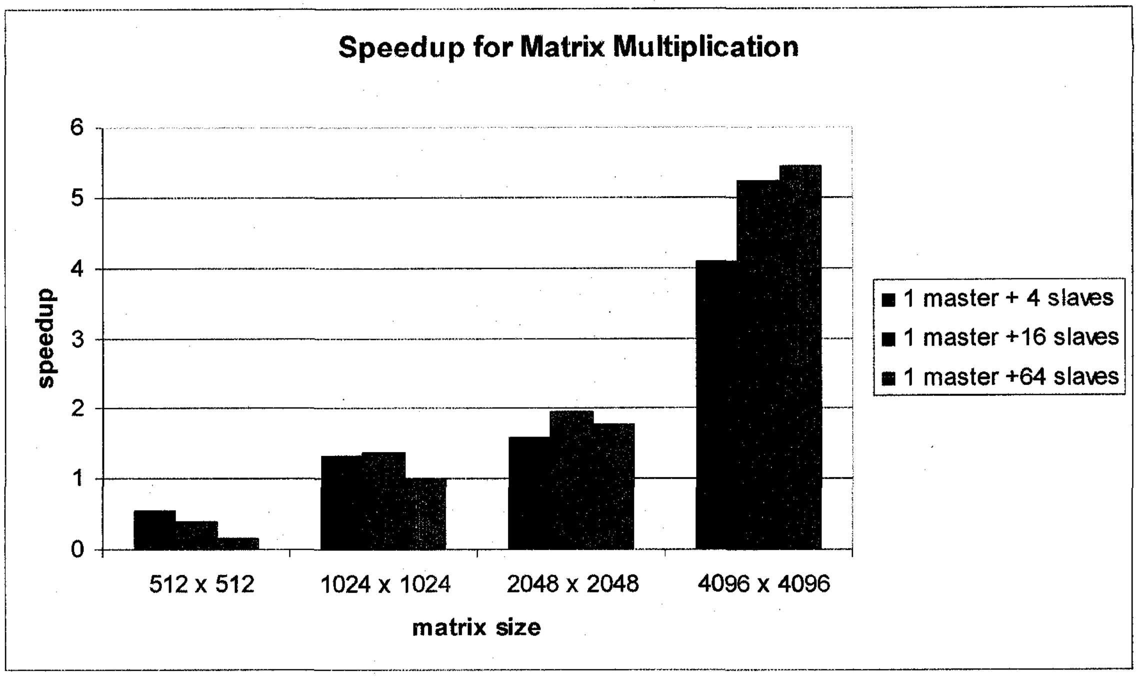


Figure 27: Speedup for Matrix Multiplication.

Contrary to the Mandelbrot computation, a linear speedup was obtained for a matrix size of 4,096 x 4,096 when using 4 processors (actually a super linear speedup of 4.089 was obtained). The super linear speedup is sometimes obtained due to the cache effect from different memory hierarchies. If the processors have a big cache memory, then more or even all core data can fit into the cache reducing the memory access time. The benefits of utilizing more processors pay off as the data size grows: for a 2,048 x 2,048 matrix size the best speedup is obtained when using 16 slaves, but for a 4,096 x 4,096 matrix size using 64 slaves gives a better speedup.

Pipeline Computation

Our active system can also be used to implement pipeline computation. The process of loaning a credit from a bank can be seen as an example of a pipeline computation.

First, we have to talk to a bank employee to explain the type of credit we want and we fill in a credit request (step1). Then we have to take the credit request to a financial clerk that would check our request, our interest and maybe credit history to determine whether we can afford the credit (step2). After the financial clerk approves the request, all the credit loans need to be additionally approved by the bank manager (step 3). Once the manager approved the loan we are done. Let us assume that step 1 takes 10 minutes, step 2 takes 20 minutes and step 3 takes 2 minutes. For one customer, it takes 32 minutes to get a loan from the bank. Once one customer is done with step 1 and moves to step 2, another customer can be helped with step 1. But still the second customer needs to wait another 32 minutes to get a loan. That means that every 32 minutes a customer gets a loan request approved (we are assuming that all loans are approved). We can improve this throughput by assigning 5 employees to help the customer with step 1 and 10 employees for step 2. This way it still takes the first customer 32 minutes to get a loan approved, but after the first customer, subsequent customers finish this loan process every 6 minutes, allowing the bank to serve more customers per day.

We have implemented this type of pipeline computation using active objects. Our example contains 5 processes (or 5 steps) that take 2, 6, 6, 8 and 2 seconds to execute (produce a result). Similar to the bank example, every 24 seconds a result is obtained.

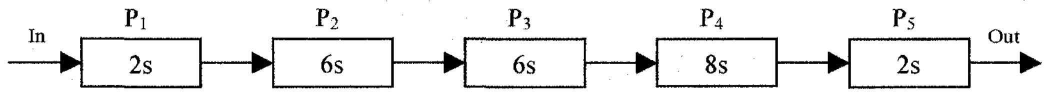


Figure 28: Regular pipeline computation with 5 processes.

This regular pipeline can be optimized to get a better throughput by adding dispersers and collectors and replicating the processes that take more time to execute. One possible arrangement of these processes is shown in Figure 29:

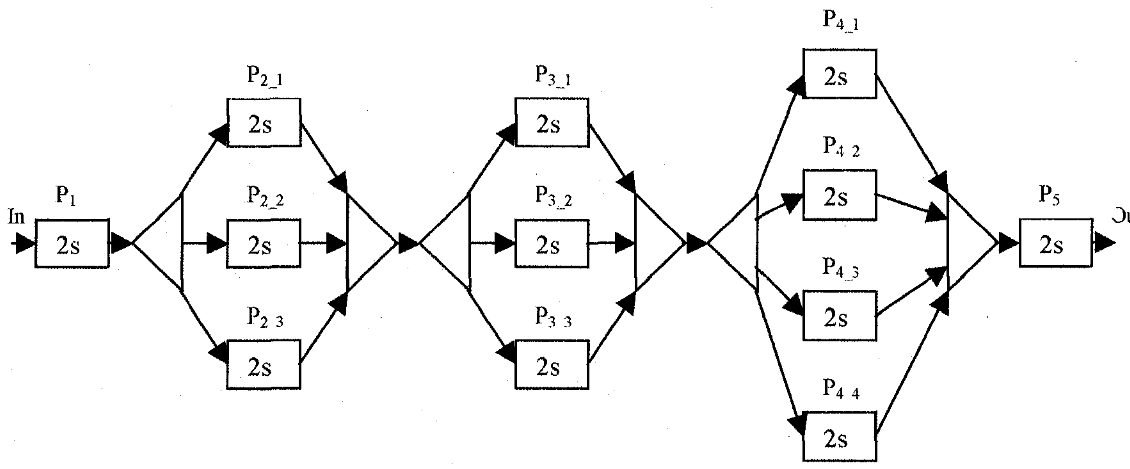


Figure 29: Pipeline computation with dispersers and collectors.

For each process that takes 6 seconds (P_2 and P_3) we replaced it with 3 similar processes: P_{2_1}, P_{2_2} and P_{2_3}. The first process (P_1) in the pipeline takes only 2 seconds to execute. Once it finishes, it sends the result to P_{2_1}. The next output of P_1 is send to P_{2_2}, and the next one to P_{2_3}. By the time P_1 produces the 4 output, P_{2_1} finished and it can process this output. Likewise, the process that takes 8 seconds (P_4) is replicated in 4 similar processes. The result of our pipeline implementation is presented in Table 3.

Table 3 Results for pipeline computation

result1	result2	result3	result4	result5	result6	result7	result8	result9	result10
10.78	10.105	10.064	10.06	10.06	10.06	10.06	10.06	10.056	10.064

The pipeline system is fed with 10 inputs every second (every second a new customer wants a loan) and the results show the output of each input. Every 10 seconds a new result is produced, instead of the initial 24 seconds. The results could have been improved if each process would take only 1 second to execute. That means that the process that takes 2 seconds would be replaced by 2 similar processes, the process that take 6 seconds with 6 similar processes and so on (see APPENDIX C for details of the implementation).

The three programs presented in this chapter demonstrate that developing parallel applications using active objects can improve the execution time of the sequential version. Not only that parts of the problem are executed in parallel, but also in a distributed manner taking advantage of the processing power of computers in the network.

CHAPTER 7

CONCLUSIONS

The Object Oriented (OO) programming model has become more popular in the last two decades as more applications were developed and designed using this model rather than the procedural or functional programming model. In the OO model, everything is considered to be an object and objects communicate with each other by sending messages. Some argue that this model is a good reflection of the real world and how objects in real world interact. Each object has state (its attributes) and behavior (its methods). However, all the objects are passive and all method invocations are executed in the caller's thread. A passive object reuses the thread of the object that created it. Assume we have an object **`John`** that creates another object, **`woodcrafter`**, that crafts chairs. When **`John`**, wants to get a chair from the **`woodcrafter`**, he sends a message **`createChair()`** to the **`woodcrafter`**. The **`woodcrafter`** is using a shared resource, a **`factory`**, to get his supplies. When the **`woodcrafter`** is blocked waiting to get the supplies from the **`factory`**, **`John`** is also blocked. Moreover the **`woodcrafter`** is using the **`John`**'s thread priority while waiting for the **`factory`** to send him the supplies he needs. The **`factory`** should execute the supply requests based on the **`woodcrafter`**'s (thread) priority and not based on **`John`**'s (thread) priority.

An active object model represents a better reflection of the real world. Each object executes the method invocations in its own thread and only sends the result to the caller.

Moreover, communication can be done both synchronously and asynchronously (in the OO model the communication is always done synchronously). Active objects queue method invocations and execute them in the order of arrival.

We have modified the Java language to incorporate active objects. This could have been done in a number of different ways: by creating an additional library [4], by creating an *Active* interface and have the active objects inherit from this interface, or by implementing patterns. We have implemented our active object system by extending the Java language with four new keywords: **`active`**, **`async`**, **`on`**, **`waitfor`** and modifying the compiler to accept these new constructs. The interaction with active objects can be both synchronous and asynchronous (when using the **`async`** keyword at the end of the method invocation). Because calls can be asynchronous (the caller continues executing while the active object executes the method), active objects can be used for developing parallel applications. Developing parallel applications in Java can be done using threads. Protecting the shared resources and avoiding seeing a resource in an inconsistent state are done by using the **`synchronized`** keyword, but it only provides a 'fake sense' of security and its overuse can lead to deadlocks. Active objects execute methods in their own thread and only one method can be executed at a time, so the object cannot 'be seen' in an inconsistent state. Unfortunately, deadlocks can still occur with active objects: i.e. when one active object calls a method on a second active object, which in turn calls a method on the first object.

Active objects do not have to reside on the same machine, but they can be distributed over the network. The location of an active object is specified when the object is created by using the extended creation expression. Active objects represent a possible model to create parallel and distributed applications. The results obtained when running applications developed with our active object system (see Chapter 6 and Appendix) are encouraging. Even though in most cases no linear speedup was obtained, the fact that some good, decent speedups were achieved demonstrates that our proof of concept is feasible.

The real world has objects that are active and objects that are passive. Both of them can be represented in our active object model, making it a better reflection of the real world.

CHAPTER 8

FUTURE WORK

Our system demonstrates one way to implement active objects in Java by extending the Java language. The system that we have created has some restrictions in how it should be used, but can be further extended or improved to eliminate some of these restrictions. The following are possible extensions of our system:

- Starting and stoping the ServerManagers from the client code: Recall that the server managers provide the remote objects that the clients will use to create active objects on the server. In the current version of our active object system, all server managers are started through an *ssh* script before running the program. The script uses a file that contains all names of the hosts on which the server manager has to be started. An alternative to this approach is to start the server manager only when an object has to be created on that server. Each active object creation on a certain host should be preceded by a verification of the state of the server manager on that host. If the manager is not running, start it and then create the active object. The starting of a manager can be included in the ClientManager's **`invokeConstructor`** method. Stopping/cleaning the server managers is currently done through an *ssh* script. Stopping/cleaning the server manager is a little more difficult, because a program stops executing in various ways: it reaches the end of the program, an unchecked exception like *NullPointerException* is thrown, or a *System.exit* method is called.

- Warning the user if an asynchronous call does not have a matching **`waitfor`**: Once a method finishes executing its body (either reaches the end of the body or executes a return statement), all the asynchronous calls that returned a result that was not waited for are discarded (the result is ignored once it gets back to the client). A possible extension may be warning the user that not all asynchronous calls with a return value have a matching **`waitfor`**. This warning is thrown by the compiler during the flow phase: every time an asynchronous method with a return value is called an index associated with that object is incremented and every time a **`waitfor`** is executed for that object, the associate index is decremented. When the method finishes executing its body (or executes a return statement), all the indexes are check and if they are not equal to zero a warning is shown to the user.
- Including an exception mechanism: The current version of our system does not forward to the client the exceptions that occurred while executing an active object's method. All these exceptions are caught in the ClientManagers and cause the execution of a program to stop. An *ActiveException* base class can be created and all active objects' methods should throw subtypes of this class. The ClientManager catches these exceptions and forwards them to the client.
- Receiving out of order invocations on the same object: Let us assume that inside a method an active object's method **`foo`** is called, followed by a call to the **`bar`** method of the same active object. Let us also assume that both **`foo`** and **`bar`**

return a result. When **`waitfor`** statement is used for that active object, the first result that is obtained is the **`foo`** return value (**`foo`** was the first method called on that active object). Our system can be extended to allow receiving out of order invocations, that is allowing the **`bar`** result to be waited first. This can be done by using a future object as a placeholder for the return value of an asynchronous call and then simply wait for that placeholder to have its value set. Instead of waiting for an active instance, the waiting is done on the future object (the future object will know the method that was invoked and the target active object).

- Keeping the active objects once they are created: Another extension could be the possibility of accessing from one application, the active objects that another application has generated, creating thus a remote object directory. In order to access an active object, a reference to it is needed. The sever manager can be modified to return a reference for a particular active object. Another possibility is that the server manager stores all the active objects in the remote registry.

APENDIX

PROGRAM CODES

A. MANDELBROT SET COMPUTATION

This section presents the parallel implementation of the Mandelbrot set computation using our active object system. The run time results of this parallel program were presented in Chapter 6. The classes used in the program are: **Mandelbrot** (represents the active object or the slave that computes the computation for a subplane), **MandelbrotClient** (represents the master and uses the **Mandelbrot** class to perform the computation on subplanes), **MandelbrotSet** and **ComplexNo** are helper classes (first is used to pass the result back to the master and the second one represents a complex number). The details of the implementation of each class are presented below.

Mandelbrot Class

```
public active class Mandelbrot {
  private static final int MAX = 256;
  //computes the Mandelbrot set for a complex plane
  public MandelbrotSet compute(Integer startLine,Integer noOfLines,
                               Integer disp_width,Integer disp_height,
                               Float real_min, Float real_max,
                               Float imag_min, Float imag_max){
    int width = disp_width.intValue();
    int height = disp_height.intValue();

    float r_min = real_min.floatValue();
    float r_max = real_max.floatValue();

    float i_min = imag_min.floatValue();
    float i_max = imag_max.floatValue();

   // compute the scale factor
   float scale_real = (r_max-r_min)/width;
   float scale_imag = (i_max-i_min)/height;

   Integer result[][] = new Integer[width][noOfLines.intValue()];
   ComplexNo c;

   //computes the Mandelbrot value for each point
   for (int y=startLine.intValue();
        y<noOfLines.intValue()+startLine.intValue();
y++){
     for (int x=0;x<width;x++){
        c = new ComplexNo(r_min+((float)x*scale_real),
i_min+((float)y*scale_imag));
        result[x][y-startLine.intValue()] = cal_pixel(c);
     }
   }
  return new MandelbrotSet(result);
 }

 public Integer cal_pixel(ComplexNo c) {
   int i=0;
   float tmp, lengthsq=0;
   ComplexNo z = new ComplexNo(0,0);
   do {
     tmp = z.real*z.real-z.imag*z.imag +c.real;
     z.imag = 2*z.real*z.imag+c.imag;
     z.real = tmp;
     lengthsq = z.real*z.real+z.imag*z.imag;
     i++;
   } while ((i<MAX)&&(lengthsq<4.0));

  return new Integer(i);
  }
}
```

ComplexNo class

```
//the class represents a complex number
public class ComplexNo {

  public float real;
  public float imag;

  public ComplexNo(float real,float imag){
    this.real = real;
    this.imag = imag;
  }
}
```

MandelbrotSet class

```
import java.io.Serializable;

public  class MandelbrotSet implements Serializable{
  private static final long serialVersionUID = -4704769797805236879L;

  private Integer[][] result;

  public MandelbrotSet(Integer[][] result){
    this.result = result;
  }
  public Integer[][] getResult() {
    return result;
  }

  public void setResult(Integer[][] result) {
    this.result = result;
  }
}
```

MandelbrotClient class

```
public class MandelbrotClient {
  private static int computerNo;
  //the computers where active objects will be created
  private static String[] computerNames;
  public int map[][];

  public int disp_width;
  public int disp_height;

  public float real_min;
  public float real_max;
  public float imag_min;
  public float imag_max;

  public String outputFile;
  public String inputFile;

  static public void main(String args[]) throws Exception {
    //get the available computer names
    computerNo = args.length-8;
    if (computerNo==0) {
      System.out.println("please specify the computers names/IPs where
                          the program runs");
    } else {
       computerNames = new String[computerNo];
       for (int i=0;i<computerNo;i++){ computerNames[i] = args[8+i]; }
         //run the client code
         startComputation(args);
      }
   }

  private void init(String args[]) throws Exception{
    disp_width = Integer.parseInt(args[0]);
    disp_height = Integer.parseInt(args[1]);

    real_min = Float.parseFloat(args[2]);
    real_max = Float.parseFloat(args[3]);
    imag_min = Float.parseFloat(args[4]);
    imag_max = Float.parseFloat(args[5]);

    inputFile = args[6];
    outputFile = args[7];
  }

  //load the required color map
```

```
public void readMapFile() throws Exception{
  FileReader fr = new FileReader(new File(inputFile));
  BufferedReader br = new BufferedReader(fr);
  map = new int[3][257];
  for (int i=0;i<257;i++){
    StringTokenizer st = new StringTokenizer(br.readLine());
    map[0][i] = Integer.parseInt(st.nextToken());
    map[1][i] = Integer.parseInt(st.nextToken());
    map[2][i] = Integer.parseInt(st.nextToken());
  }
  br.close();
}

public static void startComputation(String[] args) throws Exception{
  System.out.println("Client started!");
  long start = System.currentTimeMillis();

  MandelbrotClient mandel = new MandelbrotClient();
  //init the varibles
  mandel.init(args);

  //int how many lines of the picture should be passed to each slave
  int nrofLines = mandel.disp_height/computerNo;

  int startLine = 0;

  //create the active objects
  Mandelbrot[] mandelbrot = new Mandelbrot[computerNo];
  for (int i=0;i<computerNo;i++){
    mandelbrot[i] = new Mandelbrot() on computerNames[i];
    mandelbrot[i].compute(startLine,nrofLines,mandel.disp_width,
                          mandel.disp_height,mandel.real_min,
                          mandel.real_max, mandel.imag_min,
                          mandel.imag_max) async;
    startLine+=nrofLines;
  }

  //while the slaves compute, read the map file
  mandel.readMapFile();

  FileWriter fw = new FileWriter(mandel.outputFile);
  BufferedWriter bw = new BufferedWriter(fw);

  String l="P3\n"+mandel.disp_width+"+mandel.disp_height+"\n255\n";
  bw.write(l);

  //now wait for the results
  MandelbrotSet mandelSet=null;
  for (int i=0;i<computerNo;i++){
    waitfor mandelbrot[i]  mandelSet;

    Integer[][] result = mandelSet.getResult();
    for (int y=0;y<nrofLines;y++)
      for (int x=0;x<mandel.disp_width;x++)
        bw.write(mandel.map[0][result[x][y].intValue()]+" "
                +mandel.map[1][result[x][y].intValue()]+" "
                +mandel.map[2][result[x][y].intValue()]+" ");
      }

      bw.write("\n");
      bw.close();

      long end = System.currentTimeMillis();
      long duration = end-start;
      System.out.println("Client finished in "+duration+" millis");
     }
   }
```

B. Matrix Multiplication

This section presents the parallel implementation of the Matrix multiplication using our active object system. The run time results of this parallel program were presented in Chapter 6. The classes used in the program are: **MatrixMultObject** (represents the active object or the slave that computes the multiplication on submatrices), **MantrixMultClient** (represents the master and uses the **MatrixMultObject** class to perform the computation on submatrices) and **Matrix** (a helper class used to pass the result back to the master). The details of the implementation of each class are presented below.

Matrix class

```
import java.io.Serializable;
public class Matrix implements Serializable {
  private static final long serialVersionUID = -4476439962181063471L;
  int[][] matrix;

  public Matrix(){}

  public Matrix(int[][] matrix){
    this.matrix = matrix;
  }

  public int[][] getMatrix(){
    return matrix;
  }
}
```

MatrixMultObject class

```
public active class MatrixMultObject {

  //the neighbour form the right (with wrap around)
  MatrixMultObject right;

  //the neighbour from above (with wrap around)
  MatrixMultObject up;

  int[][] a; //the initial A submatrix for this process
  int[][] b; //the initial B submatrix for this process
  int[][] t; //submatrix A received from the neighbor after pipeing
  int[][] c; //partial sum
  int[][] newB; //submatrix B received from the neighbor after rolling
  int n; //nr of rows
  int m; //nr of columns
  Integer rank;

  int rolling = 0;

  public MatrixMultObject(int[][] a, int[][] b,Integer rank){
    this.a = a;
    this.b =b;
    this.rank = rank;

    //init c
    n = a.length;
    m = a[0].length;
    c = new int [n][m];
    this.newB = b;
    this.t = a;
  }

  public void setNeighbours(MatrixMultObject right,
                            MatrixMultObject up){
    this.right = right;
    this.up = up;
  }

  private boolean amISource(int procNr, int rank,int i){
    int sqrtProc = (int) Math.sqrt(procNr);
    int row = rank/sqrtProc;
    int col = rank%sqrtProc;

    if (col==(row+i)%sqrtProc){
      return true;
    }

  return false;
  }

  private boolean amILast(int procNr, int rank, int i) {
    int sqrtProc = (int) Math.sqrt(procNr);

    int previousStep = (i-1+sqrtProc)%sqrtProc;
    if (amISource(procNr, rank, previousStep)) {
       return true;
    } else {
       return false;
     }
  }

  public boolean startPipeing(Integer i, Integer procNr) {
    t= copyMatrix(a);
    sendRight(i,procNr);
  return true;
  }

  public boolean pipe_A(Integer i,Integer procNr){
    sendRight(i,procNr);
  return true;
  }

  public void compute(Integer step, Integer procNr){
    b = newB;

    //multiply and add:
    int i, j, k;
```

```
   for (i = 0;i < n;i++) {
     for (j = 0;j < m;j++) {
       for (k=0;k<m;k++){
         c[i][j] += t[i][k]*b[k][j];
       }
     }
   }
 }

  public boolean startRolling(Integer i, Integer procNr){
    sendUp(i,procNr);
  return true;
  }

  public boolean roll_B(Integer i, Integer procNr){
    sendUp(i,procNr);
  return true;
  }

  public Matrix getResult(){
    return new Matrix(c);
  }

  public void receiveFromLeft(int[][] t){
    this.t = t;
  }

  public void receiveFromBelow(int[][] matrix){
    this.newB = matrix;
  }

  public void sendRight(Integer i, Integer procNr){
    if (!amILast(procNr.intValue(), rank.intValue(), i.intValue())){
      right.receiveFromLeft(t) async;
      boolean finished = right.pipe_A(i,procNr);
    }
  }

  public void sendUp(Integer i, Integer procNr){
    up.receiveFromBelow(b) async;
    boolean lastRolling = amILast(procNr,rank.intValue(),i);
    if (!lastRolling) {
       boolean finished = up.roll_B(i,procNr);
    }
  }

  private int[][] copyMatrix(int[][] matrix){
    int[][] newMatrix = new int[matrix.length][];

    for (int i=0;i<matrix.length;i++){
      newMatrix[i] = new int[matrix[i].length];
      System.arraycopy(matrix[i],0,newMatrix[i],0,matrix[i].length);
    }
  return newMatrix;
  }

  public Integer getRank(){
      return rank;
    }
}
```

MatrixMultClient class

```
public class MatrixMultClient {
  public static int computerNo;
  private static String[] computerNames;

  public static int[][] readMatrix(String fileName) throws Exception{
    int[][] matrix=null;
    BufferedReader br = new BufferedReader(new FileReader(fileName));
    String str = br.readLine();
    StringTokenizer st = new StringTokenizer(str);
    int n = Integer.parseInt(st.nextToken());

    matrix = new int[n][n];

    for (int i=0;i<n;i++){
      str=br.readLine();
      st = new StringTokenizer(str);
      for (int j=0;j<n;j++){
        matrix[i][j] = Integer.parseInt(st.nextToken());
      }
    }

   return matrix;
  }

  //gets the sub matrix that is going to be send to the process number
  //send as a parameter
  private static int[][] getSubMatrix(int matrix[][],int matrixSize,
                                      int rank, int nrOfProcs ){
    int newSize = matrixSize/(int)Math.sqrt(nrOfProcs);
    int[][] subMatrix = new int[newSize][newSize];
    int row, col;
    for (int i=0;i<newSize;i++){
      for (int j=0;j<newSize;j++){
        row = i+newSize*(rank/(int)Math.sqrt(nrOfProcs));
        col = j+newSize*(rank%(int)Math.sqrt(nrOfProcs));
        subMatrix[i][j] = matrix[row][col];
      }
    }

   return subMatrix;
  }

  private static String[] readComputerNames(String file,int computerNo)
                                    throws IOException{
    File f = new File(file);
    FileReader fr = new FileReader(f);
    BufferedReader br = new BufferedReader(fr);
    String[] computers = new String[computerNo];
    for (int i=0;i<computerNo;i++) {
      computers[i] = br.readLine();
    }
  return computers;
  }

   public static void main(String args[]) throws Exception{

      if (args.length<4) {
        System.out.println("Usage: MatrixMultClient fileMatrixA
                                fileMatrixB fileMatrixC n computerNames ");
        System.exit(1);
      }

      int n = Integer.parseInt(args[3]);

      int[][] a = readMatrix(args[0]);
      int[][] b = readMatrix(args[1]);
      computerNo = Integer.parseInt(args[5]);
      computerNames = readComputerNames(args[4],computerNo);

      long start = System.currentTimeMillis();
      System.out.println("starting the computation");

      start(a,b,n,args[2]);

      long duration = System.currentTimeMillis() - start;

      System.out.println("Client finished in:"+duration+ " millis");
    }

    public static void start(int[][] a, int[][] b, int n,String fileC)
                          throws Exception{

      int[][][] subMatrixA = new int[computerNo][][];
      for(int i=0;i<computerNo;i++){
        subMatrixA[i] = getSubMatrix(a, n, i, computerNo);
      }

      int[][][] subMatrixB = new int[computerNo][][];
      for(int i=0;i<computerNo;i++){
        subMatrixB[i] = getSubMatrix(b, n, i, computerNo);
      }

      MatrixMultObject[] instances = new MatrixMultObject[computerNo];
      for(int i=0;i<computerNo;i++){
        instances[i] = new MatrixMultObject(subMatrixA[i],subMatrixB[i]
                                            ,new Integer(i))
                                            on computerNames[i];
      }

      setNeighbours(instances);

      for (int i=0;i<Math.sqrt(computerNo);i++){
        for(int j=0;j<computerNo;j++){
          if (amIPipeSource(computerNo,j+1, i)) {
            instances[j].startPipeing(new Integer(i),
                                      new Integer (computerNo)) async;
          boolean finished = false;
          waitfor instances[j] finished;
        }
      }

      for(int j=0;j<computerNo;j++){
        instances[j].compute(new Integer(i),
                             new Integer(computerNo)) async;
      }


      //only roll if it's not the last iteration
      if (i<Math.sqrt(computerNo)-1) {
        for(int j=0;j<computerNo;j++){
          if (amIPipeSource(computerNo,j+1, i)) {
            boolean ready = instances[j].startRolling(new Integer(i),
                                            new Integer(computerNo));
          }
        }
      }

    }

   //get the results
   for(int j=0;j<computerNo;j++){
     instances[j].getResult() async;
   }

   Matrix[] results = new Matrix[computerNo];
   for (int i=0;i<computerNo;i++) {
     results[i] = new Matrix();
   }

   Matrix tmp=null;
   //wait for the results
   for(int j=0;j<computerNo;j++){
     waitfor instances[j] tmp;
     results[j] = tmp;
   }

   int sqrtCompNo = (int)Math.sqrt(computerNo);
   int subMatrixSize = n/sqrtCompNo;

   //System.out.println("PRINTING THE RESULTS!!!!");
   printResult(results, subMatrixSize, n,fileC);
}

private static void printResult(Matrix[] results,int subMatrixSize,
                                int matrixSize,String fileName)
                          throws Exception{
  int sqrtCompNo = (int)Math.sqrt(computerNo);

  //create matrix
  int[][] c = new int[matrixSize][matrixSize];

  for (int i=0;i<results.length;i++){
    int row = i/sqrtCompNo;
```

```
      int col = i%sqrtCompNo;

      for (int j=0;j<subMatrixSize;j++)
        for (int k=0;k<subMatrixSize;k++)
            c[row*subMatrixSize+j][col*subMatrixSize+k]=
                                        (results[i].getMatrix())[j][k];

      }

      BufferedWriter bw = new BufferedWriter(new FileWriter(fileName));
      String str;
      bw.write(matrixSize+"\n");
      for(int i=0;i<matrixSize;i++){
        str="";
        for (int j=0;j<matrixSize;j++){
          str+=c[i][j]+" ";
        }
        str+="\n";
        bw.write(str);
      }
      bw.close();
   }

   //set the neighbors for each process
   private static void setNeighbours(MatrixMultObject[] instances){
     for (int i=0;i<computerNo;i++){
       int right = getRight(i, computerNo);
       int up = getUp(i, computerNo);
       instances[i].setNeighbours(instances[right],
                                   instances[up]) async;
     }
   }

   private static int getRight(int rank, int procNr){
     int sqrtProc = (int)Math.sqrt(procNr);
     int row = rank/ sqrtProc;
     int col = rank% sqrtProc;
     int newCol = (col+1)% sqrtProc;
    return row*sqrtProc+newCol;
   }

   private static int getUp(int rank,int procNr){
     int sqrtProc = (int)Math.sqrt(procNr);
     int row = rank/ sqrtProc;
     int col = rank% sqrtProc;
     int newRow = (row-1+sqrtProc)%sqrtProc;
    return newRow*sqrtProc+col;
   }

   private static boolean amIPipeSource(int procNr, int rank,int i){
     int j;
     int sqrtProc = (int)Math.sqrt(procNr);
     int mod = i % sqrtProc;
     int div = i / sqrtProc;

     int rankMod = (rank - 1) % sqrtProc;
     int rankDiv = (rank - 1) / sqrtProc;
    for (j = 0;j < sqrtProc;j++) {
      if (rankMod == mod && rankDiv == div) {
         return true;
       } else {
         mod = (mod + sqrtProc + 1) % sqrtProc;
         div = (div + sqrtProc + 1) % sqrtProc;
        }
    }
   return false;
}
```

C. Pipeline Computation

This section presents the parallel implementation of the Pipeline computation using our active object system. The run time results of this parallel program were presented in Chapter 6. The classes used in the program are: **Process** (represents an abstract concept of process), **Disperser** (extends **Process** and distributes the input to the available processes), **Collector** (extends **Process** and collects the output from different proceses and send them to one process), **WorkingProcess** (extends **Process** and represents the process that actually does some computation) and **Pipeline** (represents the class that feeds the pipeline with input). The details of the implementation of each class are presented below.

Process class

```
public active class Process {
  public String doProcessing(Integer packageNr){
    return "!";
  }
}
```

Disperser class

```
public active class Disperser extends Process{

  private Integer nrOfProcesses;
  private ArrayList processes = new ArrayList();
  private int index=0;
  private String computerName;

  public Disperser(Integer nrOfProcesses,Integer timeDelay,
                   Process nextProcess,String computerName){
    this.nrOfProcesses = nrOfProcesses;
    this.computerName = computerName;
    //create the collector
    Collector collector = new Collector(nextProcess) on computerName;

    //create the instances
    for(int i=0;i<nrOfProcesses.intValue();i++){
      WorkingProcess wp = new WorkingProcess(timeDelay,collector,
                                             new Boolean(false),
                                             computerName)
                                       on computerName;
      processes.add(wp);
    }
  }

  public Disperser(Integer nrOfProcesses,Integer timeDelay,
                   String computerName){
    this.nrOfProcesses = nrOfProcesses;
    this.computerName = computerName;
    //create the collector
    Collector collector = new Collector() on computerName;

    //create the instances
    for(int i=0;i<nrOfProcesses.intValue();i++){
      WorkingProcess wp = new WorkingProcess(timeDelay,collector,
                                             new Boolean(false),
                                             computerName)
                                        on computerName;
      processes.add(wp);
    }
  }


  public String doProcessing(Integer packageNr) {

    //send the request to the appropiate working process
    int i=index%nrOfProcesses.intValue();
    index++;

    WorkingProcess wp = (WorkingProcess)processes.get(i);
    wp.doProcessing(packageNr,time) async;
    String result="";
    waitfor wp result;
    result="WP"+i+" from "+computerName+" --- "+result;
   return result;
  }
}
```

Collector class

```
public active class Collector extends Process{

  private Process nextProcess;
  private int waitingForPackageNr = 0;
  private ArrayList pendingList = new ArrayList();

  private long timer;

  public Collector(Process nextProcess){
    this.nextProcess = nextProcess;
    timer = System.currentTimeMillis();
  }

  public Collector(){
    this.nextProcess = null;
    timer = System.currentTimeMillis();
  }

  public String doProcessing(Integer packageNr) {
    String result="";
    if (nextProcess!=null){
      //check if it's the right package
      if (packageNr.intValue()==waitingForPackageNr) {
        waitingForPackageNr++;
```

```
          //send the packageNr to the next process
          nextProcess.doProcessing(packageNr) async;
          waitfor nextProcess result;

          result = " collector "+result;
          //check if the next package is in the pending list
          checkPending();
        } else {
            pendingList.add(packageNr);
        }
      } else {
         long endtime = System.currentTimeMillis();
         System.out.println("S E N D I N G  O U T  T H E  P A C K A G E:"
             +packageNr+" in "+ (timer -endtime)+" milllis" );
           result = " collector ";
           timer = System.currentTimeMillis();

       }
     return result;
    }


    private void checkPending(){
      for (int i=0;i<pendingList.size();i++){
        Integer packageNr =(Integer) pendingList.get(i);
        if (packageNr.intValue()==waitingForPackageNr){
           doProcessing(packageNr);
           waitingForPackageNr++;
        }
      }
    }
  }
```

Pipeline class

```
public class Pipeline {

  private  ArrayList processList = new ArrayList();
  private ArrayList computerNames = new ArrayList();
  private int smallestDelay = 100000;

  public static void main(String args[]) throws Exception{
    if (args.length!=1) {
      System.out.println("Usage: java Pipeline inputfile");
      System.exit(1);
    }
    Pipeline pipeline = new Pipeline();
    pipeline.start(args[0]);
  }

  private void start(String inputFile) throws Exception{
    System.out.println("Client started");
    //read the nr of processes
    FileReader fr = new FileReader(inputFile);
    BufferedReader br = new BufferedReader(fr);
```

```
String processes = br.readLine();
getProcesses(processes);

String computers = br.readLine();
getComputerNames(computers);

int n= processList.size();

Process lastInstance=null;
Integer processDelay = (Integer)processList.get(n-1);
String computerName = (String)computerNames.get(n-1);

if (processDelay.intValue()/smallestDelay>1) {
   //create a disperser
   lastInstance = new Disperser(
                new Integer(processDelay.intValue()/smallestDelay),
                new Integer(smallestDelay),
                computerName) on computerName;
 } else {
   (String)computerNames.get(n-1));
  //create a working process
   lastInstance = new WorkingProcess(
                    (Integer)processList.get(n-1),
                     new Boolean(true),
                     computerName) on computerName;
 }

 for (int i=n-2;i>=0;i--) {
   processDelay = (Integer)processList.get(i);
   computerName = (String)computerNames.get(i);
   if (processDelay.intValue()/smallestDelay>1) {
     //create a disperser
     lastInstance = new Disperser(
                new Integer(processDelay.intValue()/smallestDelay),
                new Integer(smallestDelay),
                lastInstance, computerName) on computerName;
  } else {
    //create a working process
    lastInstance = new WorkingProcess(
               (Integer)processList.get(n-1),lastInstance,
                new Boolean(false),computerName) on computerName;
  }
}

System.out.println("starting the computation:");
long startTime = System.currentTimeMillis();

//start feeding the pipeline
for (int i=0;i<10;i++){
  lastInstance.doProcessing(new Integer(i)) async;
  Thread.sleep(100);
}

//now get the results
String result = "";
for (int i=0;i<10;i++){
  waitfor lastInstance result;
```

```
      System.out.println("For package :"+i + " ::: "+result+ " in "+
                          (endTime-startTime));
    }

    System.out.println("Client finished");

  }

  private void getProcesses(String processes){
    StringTokenizer st = new StringTokenizer(processes);
    while (st.hasMoreTokens()) {
      String token = st.nextToken();
      Integer delay = new Integer(token);
      processList.add(delay);
      if (smallestDelay>delay.intValue()) {
        smallestDelay = delay.intValue();
      }
    }
  }

  private void getComputerNames(String computerString){
    StringTokenizer st = new StringTokenizer(computerString);
    while (st.hasMoreTokens()) {
     String token = st.nextToken();
     computerNames.add(token);
    }
  }
}
```

BIBLIOGRAPHY

[1] R.Greg Lavender and Douglas C. Schmidt: "An object Behavioral Pattern for Concurrent Programming". http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

[2] Ben Pryor: "Implementing Active Objects with Java Dynamic Proxies". http://benpryor.com/blog/index.php?/archives/6-Implementing-Active-Objects-with-Java-Dynamic-Proxies.html

[3] Claude Petitpierrre: "Synchronous Active Objects". http://ltiwww.epfl.ch/sJava/

[4] The French National Institute for Research in Computer Science and Control. http://proactive.inria.fr/release-doc/html/index.html

[5] Wang Chen, Zhou Ying and Zhang Defu: "An Efficient Method for Expressing Active Object in C++". ACM SIGSOFT Software Engineering Notes, Volume 25, Issue 3, May 2000

[6] Carl Hewitt, "Viewing Control Structures as Patterns of Passing Messages" Artificial Intelligence, vol. 8, no. 3, June 1977, pp. 323-364.

[7] Gul A. Agha "Actos: A model of Concurrent Computation in Distributed Systems" A.I Tech Report 844, MIT, 1985

[8] Carlos A Varela and Gul A. Agha: "What after Java? From objects to actors". Computer Networks and ISDN Systems, pag 573 - 577

[9] J.E White: "High-level framework for network-based resource sharing", Internet RFC 0707

[10] Java Remote Method Invocation specifications available at: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html

[11] Message Passing Interface: http://www.mpi-forum.org/docs/docs.html

[12] Secure Shell network protocol RFC 4254: http://www.ietf.org/rfc/rfc4254.txt

[13] Java Runtime Environment: http://java.sun.com/j2se/desktopjava/jre/index.jsp

[14] Open-Source JDK: http://openjdk.java.net/

[15] Bruce Eckel: Thinking in Java, 3rd Edition, December 2002

[16] Jim Waldo, Geoff Wyant, Ann Wollrath and Sam Kendall: "A note on distributing computing". http://research.sun.com/techrep/1994/smli_tr-94-29.pdf

[17] Hesham El-Rewini and Ted G. Lewis: "Distributed and Parallel Computing", 1998

[18] Peter Welch: Communicating Processes, Components and Scaleable Systems" IFIP WG2.4, May 2001

VITA

Graduate College
University of Nevada, Las Vegas

George Oprean

Local Address:
4280 Escondido St, Ap 315
Las Vegas, Nevada, 89119

Home Address:
Septimius Severus St, Nr 19
Bloc Tol 12, Ap 3
Alba Iulia, Romania, 510136

Degrees:
Bachelor of Science, Computer Science, 2005
Technical University, Cluj Napoca

Thesis Title: An Implementation of Active Objects in Java

Thesis Examination Committee:
Chairperson, Dr Jan 'Matt' Pedersen, Ph. D.
Committee Member, Dr. Thomas Nartker, Ph. D.
Committee Member, Dr. David Pinelle, Ph. D.
Graduate Faculty Representative, Dr. Jacimaria R. Batista, Ph. D.