Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Asynchronous Active Objects in Java

George OPREAN, Jan B. PEDERSEN

School of Computer Science, University of Nevada

opreang@unlv.nevada.edu, matt@cs.unlv.edu

Abstract. Object Oriented languages have increased in popularity over the last two decades. The OO paradigm claims to model the way objects interact in the real world. All objects in the OO model are passive and all methods are executed synchronously in the thread of the caller. Active objects execute their methods in their own threads. The active object queues method invocations and executes them one at a time. Method invocations do not overlap, thus the object cannot be put into or seen to be in an inconsistent state. We propose an active object system implemented by extending the Java language with four new keywords: active, async, on and waitfor. We have modified Sun's open-source compiler to accept the new keywords and to translate them to regular Java code during desugaring phase. We achieve this through the use of RMI, which as a side effect, allows us to utilise a cluster of work stations to perform distributed computing.

Keywords. Active Objects, Asynchronous Active Objects, Java, distributed computing.

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 the same object, many different threads can invoke methods at the same time, possibly leaving the state of the object inconsistent. Using the *synchronized* keyword only provides a fake sense of security [1] 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.

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. 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 execute the code of the requested method in its own thread of control, thus excluding other threads from manipulating its data and executing its methods.

Active Objects and the Real World

The argument that "objects are considered harmful" has been borrowed from the process oriented design community [1]. 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 again 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 (or in *friend*'s 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 certainly 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) [2] or in OO terms Remote Method Invocation (RMI) [3], which both involve transferring method parameters and results back and forth between the calling process (client) and the remote object (server).

An active object system can be mimicked, and ultimately ours is indeed 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.

The active object model has been the subject of intensive research since late 80s and now the active objects are used in diverse areas: query processing [4], building concurrent compiler [5], implementing services in Telecommunication Management Services [6], developing applications for smartphones [7,8] or developing collaborating games [9].

Asynchronous Active Objects

Our active object system can easily 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, synchronous. This means that the caller blocks until the remote method invocation returns.

A well known technique for parallelising computations is using message passing; the program is decomposed into a number of processes whose only way to synchronise is through communication using message passing. Messages must be explicitly sent and received. This can be done both synchronously or 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 thereafter. When the return value is needed, a new **waitfor** statement is executed. If the value arrives before this statement is executed, the effect of the **waitfor** statement is a binding of the returned value to the variable without blocking. If the return value is not yet present, the statement will block the execution until the value is available. Another way to implement asynchronous active objects is with the use of a future object, that acts as a placeholder for the returned result [10,11].

Moreover, we modified the object creation expression syntax to allow the programmer to specify the computer on which the active object will reside. This is similar to spawning a process with MPI_spawn. Thus we have obtained a system that can be used for developing parallel and distributed applications using asynchronous active objects.

As a desirable side effect, the model allows us to write parallel code that uses asynchronous active objects. In section 6.1 we describe three different distributed computations that we implemented using asynchronous active objects.

1. Related Work

The first time the concept of active objects (actors) was presented was Hewitt's actor model in 1977 [12]. In his model, the actor was the central entity that executed its actions: communicates with other actors, creates actors and changes its behaviour. The location of the actors can be distributed and they can execute actions in parallel. Later a mathematical definition for the behaviour of an actor system was presented [13]. This actor (active object) model is suited perfectly for creating parallel and distributed applications. Unfortunately the actor model was not as popular as procedural, functional or OO programming. The parallelism model offered by the procedural or OO model is not as powerful as the actor model, so a number of ways to integrate active objects into the OO paradigm have been proposed.

Java RMI proposed a solution for accessing remote (distributed) objects in the same manner as accessing local objects. The location of the object would be transparent to the programmer, using the same '.' operator to invoke methods on local or remote objects. While making the process of developing distributed applications easier, the RMI (and RPC) model is a synchronous client/server model. The client has to wait for the response from the server before it can resume its execution. Better performance can be obtained if the client can use this "waiting time" to do other operations that do not involve using the return value from the server. If we consider a real world example, when someone wants to prepare breakfast for her family and realizes that she does not have milk for the cereal, she can ask somebody (an active object) to get some cereal (the result of the method call) from the store. During the time she gets the cereal (is waiting for the result), she 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 the cereal: get the milk from the refrigerator, get the cereal bowls from the cabinet, set the spoons on the table and maybe pour orange juice in some glasses. Once she gets the cereal, breakfast can be served.

Active objects have been the subject of intensive research since mid to late 80s. Issues like garbage collection of active objects [14,15,16,17,18], exception handling [19,11], type theory of active objects [20,21,22], transition from active objects to autonomous agents [23,24] and many more have been addressed. As a result of this research, a number of different ways to integrate concurrency into object-oriented languages, using active objects, exist. These approaches are categorised in [25]: the library approach (create new libraries for concurrent and distributed programming), the integrative approach (extend the language, rather than the library), and the reflective approach (that is a "bridge" between the two previous approaches).

We have opted for the integrative approach, that consists of merging concepts of object and process/activity, thus creating the new concept of active object.

An Active Object pattern can be used to implement active objects. This pattern decouples method execution from method invocation to enhance concurrency. A number of components like Proxy, Scheduler, Servant, Activation Queue, Method Request and Future have to be implemented for this pattern to work [10]. The programmer does not only have to concentrate on implementing the actual active object, but also understand how these components interact and then implement them. Java Dynamic Proxies offers another solution to implementing the active objects [26].

Claude Petitpierre integrated active objects in the Java language by adding new keywords [27]. His approach uses the MPI model of sending and receiving messages: one object calls a method on a second object (ready to send the message) and the second object has to accept that method (ready to receive the message). The keywords added are: *active*, *accept*, *select* and *waituntil*. His approach models only synchronous active objects. The C++ language has also been extended, with *active* and *passive* keywords, to integrate active objects. One example by Chen et al. is [28]. The implementation of active objects has a transaction service to maintain the atomicity of a 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.

Creating an external library that can be integrated with the Java language (similar to the MPI library [29] for C and Fortran, and MPJ (A message passing library) for Java [30]) is another possibility of having active objects in the OO model. The French National Institute for Research in Computer Science and Control has created such a library called ProActive [11]. Regular passive objects can be turned into active objects, asynchronous method invocation is supported and a future object is used as a placeholder for the actual result. Restrictions are imposed when using this library: final methods cannot be used in active objects, final classes cannot be used to instantiate active objects, and the programmer cannot override functions like *hashCode()* or *equals()*. Moreover, the use of the *toString()* method with future object can lead to deadlock.

Another Java library that can be used for developing concurrent systems is *Communicating Sequential Processes for Java* (JCSP) [31,32,33]. It uses Hoare's algebra of Communicating Sequential Processes (CSP), a mathematical model for concurrency that can guarantee (and prove) that the multithreaded application developed with it does not have deadlock, livelocks, race hazards, or resource starvation. JCSP views the world as layered networks of communicating processes. The processes interact via channels (or other CSP synchronisation objects like barriers or shared-memory CREW locks) and not via method invocation (our active objects communicate with each other by sending messages through RMI). Similar to our active object system, JSCP offers the programmer an alternative to developing concurrent application without the use of synchronized, wait, notify or notifyAll primitives (the JCSP actually uses internally these primitives and so does our active object system). The JCSP.net [34] (an extension of JCSP for networking) allows processes to be distributed on the network. Objects sent across JCSP.net channels are serialised. These channels can be set to allow automatically download the class if it does not exist on the receiving side (we have used RMI for our system for this purpose).

Java is not the only object-oriented language that has libraries that can be utilised when developing applications that involve active objects. DisC++ [35] and ACT++2.0 [36], are two libraries for concurrent programming in C++.

A number of other, less mainstream, but nevertheless exciting, languages that support active objects exist: Cω, a research language based on the join calculus [37], which is an extension of C# for asynchronous wide-area concurrency [38]. Active objects in Cω are supported through the techniques of using chords (A chord is a method with multiple headers, all providing parameters for the body, and all but one being called asynchronously); ACTALK is a framework integrated with Smalltalk programming environment [39]; Hybrid is an object-oriented language where objects are active entities [40]; Correlate is a concurrent object-oriented language that supports active objects as a unit of concurrency [41]; TCOZ is an extension of Object-Z with timed CSP and timing constructs [42].

2. Implementation Choices

A number of important choices must be made when embarking on language development. Some of the most important ones are as follows:

- Should we chose to develop an brand new language or provide an extension to an existing language? Thousands of very exciting languages exist, all developed with one thing in mind: solving particular problems that the developer feels are not handled well by existing languages. However, convincing people to adopt new languages seems to be a daunting task. We decided to extend Java, a language that has been accepted in almost all areas of computing.
- How should the new functionality be provided? Since we are using Java, two distinct choices are available. First: extend than language with new keywords, and amend the compiler or provide a pre-processor. (This was the original approach taken by the developers of C++). Second: provide a library of (precompiled) classes that implements the functionality (JCSP uses this approach). We chose the first approach; providing new keywords and amending the compiler; A down side of this approach is of course the difficulty in maintaining the compiler when a new version of the language comes out, but we believe that the learning curve of learning to write concurrent programs with new keywords/syntax compared to the library approach, is less steep, and the code looks cleaner. Also, we believe that this enables novice programmers to more easily pick up the concept of active objects.
- Should the implementation allow for remote active objects or only local ones? If we only allow local active objects, data sharing can be realized through the use of the standard Java primitives like the `protected` keyword and the use of locks and monitors. This is not possible if we wish to allow for remote active objects.
- To support remote active objects, we need to decide how networked data transfer should be handled. Again, two different approaches exist; first: implement data exchange through TCP/UDP sockets; second: use a library like mpiJava [43], which implements a messaging system though TCP, or three: use the Java provided RMI functionality. Implementing messaging directly using TCP is the most speed efficient method, but we decided to use RMI for a number of reasons: It is already provided in Java, but more importantly, the RMI system provided by Java automatically allows for classes not already on the remote system to be downloaded through a web service.

This implementation of asynchronous active objects in Java is a prototype that in time might reveal points in its implementation that need to be optimised, especially if serious parallel computing is to be done using an asynchronous active object system. However, we believe that the choices we have made serve as a good basis for an asynchronous active object system integrated in into Java.

3. Asynchronous Active Objects in Java

We have implemented our asynchronous active object system in Java, by extending the language to integrate both synchronous and asynchronous active objects; we chose Java for a number of reasons:

- The language is already object oriented.
- It supports reflection.
- It has RMI 'built in'.
- The Java compiler 1.6 is available as open-source.
- Auto-boxing is done automatically (since Java 1.5).

- It is platform independent.

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

- It must be *active*, that is, it executes the methods in its own thread.
- It must be possible to place an active object on any machine reachable on the network that supports Secure Shell scripting (ssh) [44] and the Java Runtime Environment (JRE) [45].
- Method invocation can be synchronous or asynchronous.
- A way to obtain the result of an asynchronous invocation must exist.

3.1. New Java Keywords

We decided to add new keywords to Java 1.6, and then utilise the Java compiler to parse and compile them. This addition consists of 4 new keywords/constructs:

- A new **active** modifier, which can only be placed on a class declaration.
- An extended object creation expression:

  new *<active class>(...)* **on** *"machine name";*

  which creates an active object instance of *<active class>* on machine *"machine name"*. The last part (**on** *"machine name"* is optional, and if left out the object will be created on the local machine.
- An extended method invocation expression:

  <active object>.<method>(...) **async***;*

  The **async** keyword makes the method invocation asynchronous, that is, the control returns immediately.
- A new blocking **waitfor** statement:

  waitfor *<active object> <variable>;*

  This causes the execution of the thread to be temporarily suspended until the asynchronous method invocation has returned a value into the variable.

3.2. Restrictions on Using the New Keywords

The design of our active object system restricts the usage of the new keywords or constructs:

- The new creation expression can only be used to create active objects. Passive objects use the regular syntax.
- The **async** keyword can only be used after method invocations. If multiple method calls are chained in the same expression, **async** only applies to the last method invocation:

  activeObj.foo().bar() **async**

  the *bar* method will be called asynchronously.
- Logically, asynchronous method invocation may only used as an expression *statement* (where its return value, if not *void*, is not immediately needed). So, it cannot be used on the right RHS of an assignment or as an argument in another method invocation.
- Passing *null* as an argument of a method of an active object is not allowed. Reflection is used to determine which method needs to be invoked. Because of method overloading, the name of the method is not enough and the parameter types are used to select the method that needs to be invoked. *null* has no type, so the decision can not be made.

- Waiting for the results of an asynchronous invocation on the same active instance has to be done in the same order as the invocation. If *foo* and *bar*, in this order, are invoked on the same active object, then first the result of *foo* has to be waited for and then the result of *bar*.

4. Implementation

In the OO model, an object sends a message to another object (invoking a method is sometimes erroneously described as sending a message) synchronously. We have extended this model of interaction by allowing objects to "communicate" both synchronously and asynchronously. The active object executes the method invocations in its own thread and has a queue of pending messages that must be processed. An active object can be created on any reachable machine from the network that supports Secure Shell (ssh). The communication with active objects is done by exchanging real messages and is realized through RMI.

Let us assume that the machine creating the active object is called *client* and the machine where the active object resides is called *server*. The *client* can send the *server* two types of messages:

- **create message** asking the *server* to create a new active object.
- **invoke message** asking the *server* to execute a method on one of the active objects that it hosts.

4.1. Creating an Active Object

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

activeObj = new ActiveClass(args) **on** *"server"*

where again, the **on** *"machine"* part is optional, and if omitted, the object will reside one the local machine. The creation of an active object happens synchronously and the *server* replies with another message sending the client a handle to the active object, as shown in Figure 1.

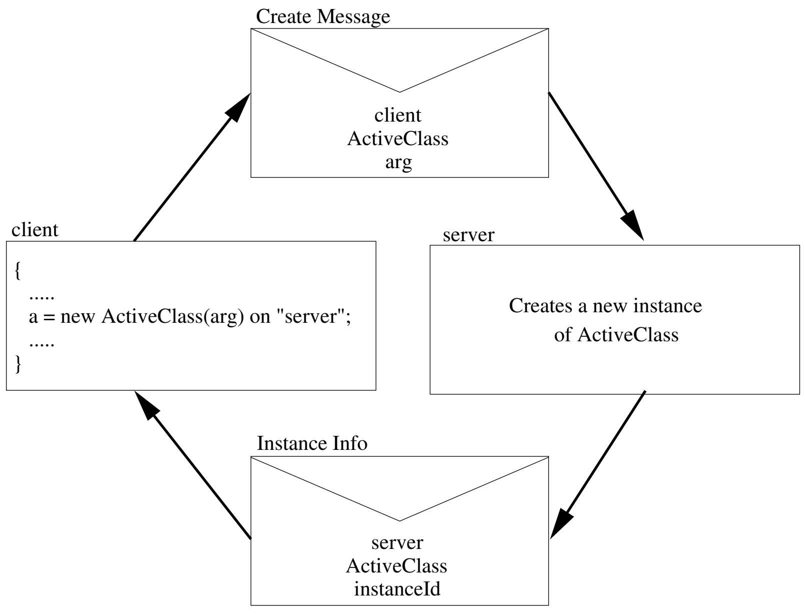

Figure 1. Active object creation.

The **create message** contains: the name of the machine that initiated the call (used internally by the server to keep track of invocations from each client since active object references can be passed from machine to machine), the type of the class, and the arguments of the constructor. The *server* sends the *client* a 'remote reference' represented by an *InstanceInfo* object that contains the following information: the name of the machine where the object resides, the type of the object and a unique identifier to distinguish this instance from other instances on the same *server*. The *client* then uses this remote reference every time it wants to send a message to the active object.

4.2. Invoking Methods on an Active Object

Unlike passive objects, the methods of active objects can be invoked asynchronously:

activeObj.foo(a,b,c) **async**

The *client* sends an **invoke message** asking the *server* to execute the *foo* method on the *activeObj* and return the result asynchronously. The *client* uses the 'remote reference' sent by the server, as shown in Figure 2.

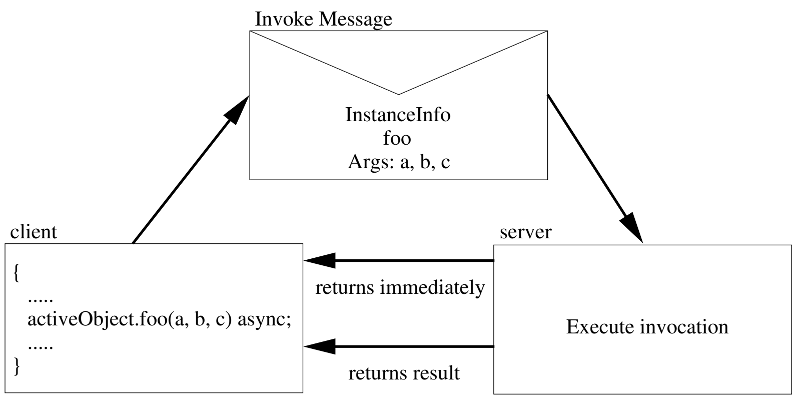

Figure 2. Active object invocation.

The call returns immediately since the **async** keyword is used and the client continues executing the rest of the code. We have implemented the asynchronous invocation by creating a thread that performs the actual call. This thread uses RMI to invoke the method on the active object. This RMI call is synchronous, but the main thread continues with the rest of the client code, thus simulating asynchronous behaviour. When the *server* finishes executing the method, the thread that carried out the execution signals the main thread that the result is available. If a method is invoked synchronously on an active object, then the main thread blocks and waits for the result, unless the method returns void. In this case the invocation is sent to the server and the main thread continues its execution.

4.3. Waiting for 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:

waitfor *activeObj var ;*

From the programmer's point of view, the **waitfor** statement can be interpreted 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*". The **waitfor** statement has to be used only for asynchronous methods that return a value. If two methods that return a value, *foo* and *bar* in this order, are invoked on the same active object, then the return values have to be waited in the order of invocation, first the result of *foo* and then the result of *bar*. The compiler will not complain if the the order of waiting for the result is reversed, but a *ClassCastException* will be thrown at runtime if the types of the two return values are not compatible.

The programmer may forget (or choose not to) to wait for the result of an asynchronous call. In this case, when a method finishes executing its body, all the results of the asynchronous invocations that were not waited for are discarded/ignored. This implies that all the results of asynchronous calls have to be waited for in the same method in which they were made.

4.4. Message Ordering

An active object accepts requests from any machine that has a reference to it. Active objects can be passed around, so not only the machine that created the object can have a reference to it. The active object will execute the requests in the order of arrival, on a first come, first served basis. No ordering can be imposed on invocations from different machines on the same active objects, in accordance with Lamport's happen-before relation [46]. Though, if two invocations, *foo* and *bar* in this order, are invoked on the same active object, then the *foo* will be executed before *bar*.

4.5. ClientManager and ServerManager

The core components of our active object system comprises two classes: *ClientManager* and *ServerManager*. The communication between the machine that creates the active object and the machine that hosts the active object is realized through these classes. The creation of an active object or a method invocation of an active object is translated at compile time to an invocation to one of the following *ClientManager* methods:

- *invokeConstructor(...)* creates an active object and returns a 'remote reference' of type *InstanceInfo*. This remote reference is used by the *client* every time it wants to invoke a method on the active object.
- *invokeMethod(...)* invokes a method on an active object.

The *ClientManager* keeps track of all the active object invocations that are initiated from the machine on which it runs. As some of these invocations are executed asynchronously, the return values of these calls are also managed by the *ClientManager*. Besides the two core functions, the *ClientManager* also has additional helper methods:

- *getMethodId(...)* returns a unique identifier that is associated with every method that has an active invocation in its body. It is inserted by the compiler at the beginning of each method body that has at least one active object invocation.
- *removeUnwaitedCalls(...)* removes all the un-waited calls within a method body. It is inserted at the end of all methods that have at least one active object invocation.

The *ServerManager* plays a similar role on the server side (a server is any machine that hosts an active object). It accepts **create** and **invoke** messages and sends these requests to the corresponding active object. There is only one *ServerManager* on each machine that participates in the execution and it has to be started before the program is executed. Starting the managers is done by executing an ssh script.

4.6. Compiler Modifications

The programmer does not know about the *ClientManager* and *ServerManager* classes, but instead he writes Java code using the added keywords. At compile time, during desugaring, the new syntax is replaced by calls to *ClientManager* methods.

The modified compiler performs two important tasks: first it checks if the syntax is correct and that the new keywords are used properly and secondly, during the desugaring phase, it replaces the new syntax with regular Java code.

The **active** keyword is removed from the class declaration. During the attribution phase additional information is saved in the node that represents a class to identify an active class.

The generation of the *methodId*, the translation of the new creation expression, asynchronous invocation and **waitfor** statement to regular Java code will be shown in the Mandelbrot example (section 6.1).

5. Example

To demonstrate the use of the asynchronous active object system we have implemented a simple publisher/subscriber system; to be exact, we have implemented the example given in the Cω tutorial [47]. The Cω implementation implements 3 classes: *ActiveClass*, *Subscriber*, and *Distributor*; in addition it implements an interface called *eventSink*. Since we have the notion of active objects built into the language and the compiler, we do not need to define a class for an active object, and thus need only classes for the *Subscriber* and the *Distributor*, as well as the driver with the main method (Figure 3).

The main method creates a distributor and 3 subscribers, and adds the subscribers to the distributor, who in time invokes the `post` method of these; subsequently the subscribers print out the message received.

In this example we do not make use of the ability to create *remote* active objects; if the optional **on** *"machine"* part of the new object creation is left out, then the new object will be created on the local host. See section 4.1.

The output obtained (one of the many possible) when executing the code is:

```
a got the message: First message
b got the message: Second message
b got the message: Third message
c got the message: Third message
a got the message: Second message
a got the message: Third message
```

Note, there exist no total ordering for all the messages, but a total ordering within the each subscriber does exist. This is a sided effect of the message ordering constraints that we implemented (See section 4.4), something which the Cω implementation does not guarantee.

In the next section we describe another desirable side effect of our asynchronous active object system; namely the ability to implement parallel computation using active objects.

6. Active Objects for Distributed Computing

By utilising asynchronous method invocation, our system allows parallel computation in a very simple manner. Moreover, the active objects do not have to be located on the same machine, but can be distributed over the network. The distributed support offered by Java is RMI. By calling methods on active objects synchronously we obtain the same effect as RMI without searching the registry for the remote object. By calling methods asynchronously (add the **async** keyword at the end of the invocation) and by allowing the caller to continue

```
public active class Distributer {
    private ArrayList<Subscriber> subscribers = new ArrayList<Subscriber>();

    public void subscribe(Subscriber s){
        subscribers.add(s);
    }
    public void post(String message){
        for (Subscriber s:subscribers){
            s.post(message) async;
        }
    }
}

public active class Subscriber {
    private String name;

    public Subscriber(String name) {
        this.name = name;
    }
    public void post(String message){
        System.out.println(name+" got the message:"+message);
    }
}

public class Demo {
    public static void main(String args[]){
        Distributer d = new Distributer();
        Subscriber a = new Subscriber("a");
        d.subscribe(a) async;
        d.post("First message");
        Subscriber b = new Subscriber("b");
        d.subscribe(b) async;
        d.post ("Second message");
        Subscriber c = new Subscriber("c");
        d.subscribe(c) async;
        d.post("Third message");
    }
}
```

Figure 3. The active *Distributor* and *Subscriber* and passive driver class.

his execution without waiting for the result at the invocation time, we obtained a parallel computing model.

6.1. Mandelbrot Example

One often used pattern in distributed systems is the master/slave architecture. The master divides the work into tasks and sends the tasks to the slaves. The slaves execute the task and send the result back to the master. We will show how this master/slave model can be designed with our system by implementing the Mandelbrot Set computation. The active code for this implementation can be seen in Figure 4 (see lines 1, 11, 12, and 16 for the use of these new keywords). Executing our compiler with the code from Figure 4 will generate the Java code shown in Figure 5.

The first and the last statements of the *main* method of the *MandelbrotClient* class were added by our compiler. The *methodId* (line 9) is used by the *ClientManager* to handle method invocations and the last statement (line 26) discards all the results of any asynchronous invo-

```
public active class Mandelbrot {
    public MandelBrotSet compute(...) {
      ... compute the mandelbrot set for the area given by the parameters
    }
}

public class MandelbrotClient {
    public static void main(String[] args) {
        Mandelbrot[ ] mandelbrot = new Mandelbrot[noProc];
        for(int i=0; i<noProc; i++) {                                          10
            mandelbrot[i] = new Mandelbrot() on computerNames[i];
            mandelbrot[i].compute(...) async;
        }
        MandebrotSet mandelSet;
        for(int i=0; i<noProc; i++) {
            waitfor mandelbrot[i] mandelSet;
            ... process the return value
        }
    }
}                                                                              20
```

Figure 4. Mandelbrot fractal computation using asynchronous active objects.

```
public class Mandelbrot {
    public MandelBrotSet compute(...) {
      ... compute the mandebrot set for the area given by the parameters
    }
}

public class MandelbrotClient {
    public static void main(String args) {
        Long methodId = ClientManager.getMethodId();
        InstanceInfo[ ] mandelbrot = new InstanceInfo[noProc];                 10

        for(int i=0; i<noProc; i++) {
            mandelbrot[i] = ClientManager.invokeConstructor(
                "Mandelbrot", new Object[ ]{}, computerNames[i]);
            ClientManager.invokeMethod(
                methodId, mandelbrot[i], "compute", new Object[ ]{....}, true, false);
        }

        MandebrotSet mandelSet;
        for(int i=0; i<noProc; i++) {                                          20
             ReturnObject returnObject0 = ClientManager
                  .waitForThread(methodId, mandelbrot[i], false);
             mandelSet = (MandelbrotSet)returnObject0.getReturnValue();
             ... process the return value
        }
        ClientManager.removeUnwaitedCalls(methodId);
    }
}
```

Figure 5. Mandelbrot code generated by our compiler.

cations from this method body that were not waited for. The construction of an active object (line 11 in Figure 4) is translated to a call to *invokeConstructor* (line 13-14 in Figure 5) and the active object method invocation (line 13 in Figure 4) is translated to a *invokeMethod* call (lines 15-16 in Figure 5). Finally the **waitfor** (line 16 in Figure 4) statement is translated into two statements: first a call to *waitForThread* to wait for the result to be available and secondly code for getting the actual result.

6.2. Matrix Multiplication Example

We have also implemented Fox's Pipe-and-Roll matrix multiplication algorithm [48] with active objects. The algorithm divides the matrices, A and B, into sub matrices in a grid manner and then the following steps are repeated: broadcast A (the first matrix) to the right, do local multiplication of incoming A and local B (second matrix), and shift B up with wraparound. An active object (slave) receives an A sub matrix and a B sub matrix and performs the three steps from the algorithm. A master object controls the number of iteration for the algorithm and on each iteration it sends the slaves the task they have to perform. The results obtained after running our example is presented in section 7.2.

6.3. Pipeline Computation Example

Another example that we have implemented with our active object system is a pipelined computation. Our example contains 5 processes (or 5 steps) that take 2, 6, 6, 8, and 2 seconds respectively to execute (to produce a result at the end of the pipe). The throughput is no better than one result ever 8 seconds, and the latency is no better than 24 seconds. This pipeline is illustrated in Figure 6.

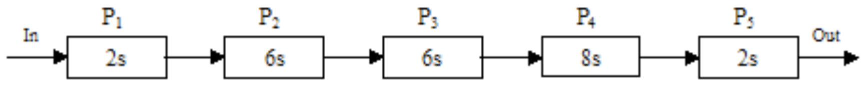

Figure 6. Original Pipeline.

As stated, we cannot get a better throughput than one result every 8 seconds. This can be optimised to get a better throughput by adding dispersers and collectors, and by replicating the processes that take more time to execute. One possible arrangement of these processes is shown in Figure 7.

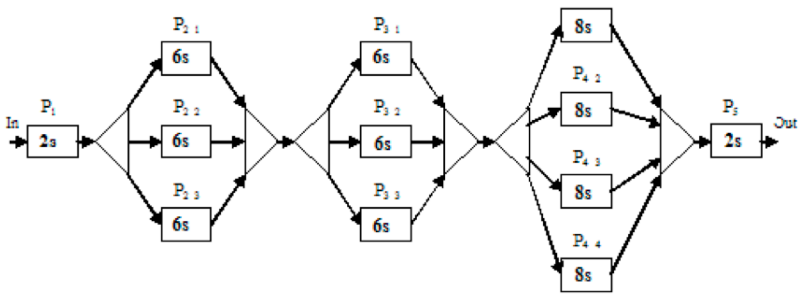

Figure 7. Optimised Pipeline with Dispersers and Collectors.

The dispersers, the collectors and the replicated processes are all active objects. The disperser creates the necessary number of processes and the collector. Once a process finishes its work, it forwards the result to the collector, which in turn forwards it to the next process or disperser. The results of the pipeline computation are presented in section 7.3.

With this new setup, we should be able to improve the throughput to one result every (no better than) 2 seconds. Naturally there might be a little overhead from passing data through dispersers and collectors.

7. Results

Although the main focus of this paper is the implementation of an active object system in Java, we do wish to present the initial results of the experiments of *parallel programming with active objects*.

7.1. Mandelbrot Set

We have tested the Mandelbrot Set computation with active objects against the sequential Mandelbrot. We started 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 each complex plane size. The speedups obtained are presented in Figure 8.

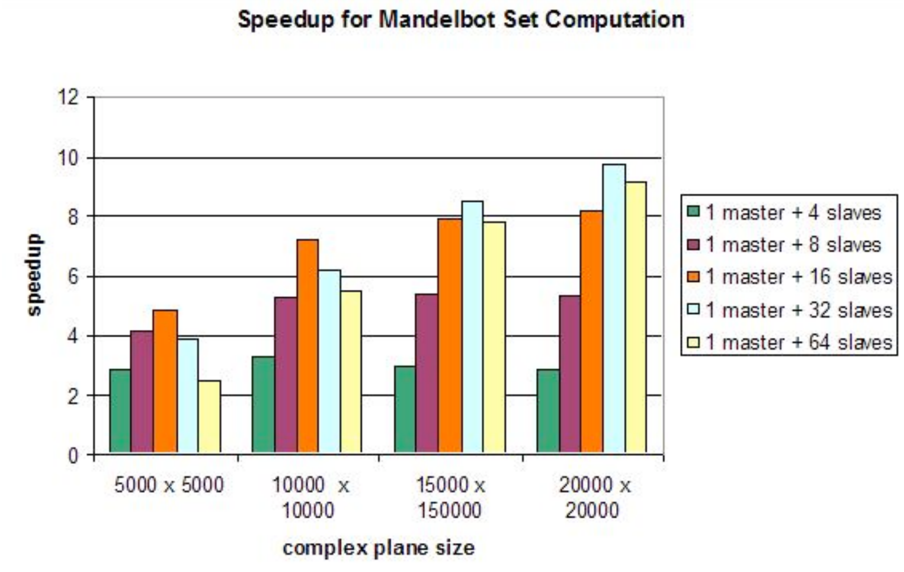

Figure 8. Speedup for the Mandelbrot Computation.

7.2. Matrix Multiplication Results

We have tested the Matrix Multiplication for matrix sizes of 512 by 512, 1,024 by 1,024, 2,048 by 2,048 and 4,086 by 4,086 on 4, 16 and 64 processors (slaves). The benefits of distributing the workload on multiple computers and doing the computation in parallel starts paying off for the matrix size 2,048 by 2,048 and 16 processors. There is some overhead of starting each slave (active object) and sending the parameters across the network (in each step, each active object “pipes” its A sub matrix and “rolls” is B matrix). The speedups obtained are presented in Figure 9.

7.3. Pipeline Computation Results

The regular pipeline example presented in section 6.3 creates a result each 24 seconds. By interspersing dispersers and collectors, and at the same time replicate the processes in the stages that take longer, the optimised version improves the throughput to one result every 2.008 seconds on average (2 is the minimum we could ever hope to achieve).

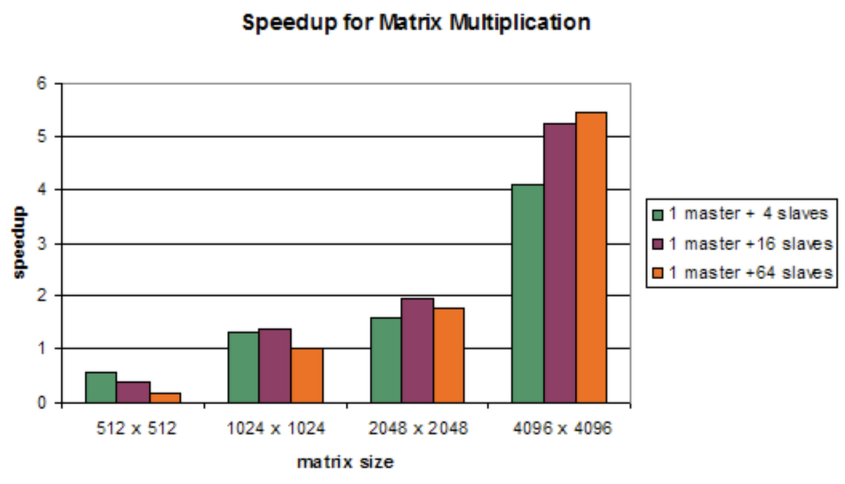

Figure 9. Speedup for the Matrix Multiplication algorithm.

8. Conclusion

In the standard Object Oriented (OO) model everything is considered an object. Objects communicate with each other by sending messages. Some argue that this model is a good reflection of the real world and of how objects interact. However, all the objects are passive and all the method invocations are executed in the caller's thread. A passive object reuses the thread of the object that created it.

The active object model represents a better reflection of how objects in the real world interact. Each object executes the method invocations in its own thread and only sends the result to the caller. Moreover, the communication can be done both synchronously and asynchronously (in the OO model the communication is always synchronous).

We have modified the Java language to integrate active objects by adding four new keywords: **active**, **async**, **on** and **waitfor**. Because methods can be invoked asynchronously using the **async** keyword, active objects can be used for developing parallel applications (the caller does not wait for the result at the moment of the invocation but continues its execution in parallel with the active object invocation). 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 and it can not be changed afterward, even though active object references can be passed around. Our system allows the existence of both passive and active objects, just like the real world. We used the active object model to implement a number of applications: Mandelbrot set computation, pipe-and-roll matrix multiplication, and pipeline computation. Even though linear speedup was not achieved, the results are encouraging and demonstrate the feasibility of using active objects in Java for certain types of distributed computation.

In general we we have shown one way of implementing asynchronous active objects in Java, using RMI and compiler modifications, which provides an easily approachable introduction to active objects for Java programmers.

Finally, since current versions of the Java Virtual Machine (JVM) are implemented to take advantage of multi-core processors, our active object system provides a straight forward way to write code for such architectures. As long as the work load associated with executing a method (invoked asynchronously) is significant, the advantage of utilising multiple cores will outweigh the overhead.

9. Future Work

Our system demonstrates one way to implement active objects in Java by extending the language. The system that we have created enforces some restrictions of how the new keywords should be used, but can be further extended and improved to eliminate some of these restrictions. Some improvements are:

- starting/stopping the *ServerManager* from code instead of ssh script
- warning the user if not all asynchronous calls with return value have a matching **waitfor** statement
- creating an exception handling mechanism (*ClientManager* catches all the exceptions sent by the active objects and does not forward them to the caller)
- keeping the active objects once they are created and allowing them to be accessed by other applications (creating a remote directory)
- Rather than creating a new thread for each asynchronous method invocation and then discarding it after the RMI call returns, a pool of threads can be kept, these can be reused.
- As illustrated in the Mandelbrot example, the **waitfor**s are handled in the same order as the invocations, this could be a potential slowdown. A possible solution to this could be a method for doing a number of **waitfor**s in parallel.
- If an active object resides on the local machine, the use of RMI will be a significant overhead, and totally unnecessary; an obvious optimisation would be to simply exchange messages within the shared memory space using locks and monitors.

One interesting, and important issue of using active objects is that of deadlocks. It is not hard to write a program that causes a deadlock; recursive calls to active objects, mutually recursive calls between active objects are simple examples that will certainly cause a deadlock. Like many other programming models, there are certain difficulties that cannot automatically be avoided. However, it it possible to observe and report such deadlocked situations; by logging all calls and keeping an up-to-date call graph in a central manager process, deadlocks can be reported to the programmer at run-time. This is an extension that we would like to add to the active object system in the future.

References

[1] Peter H. Welch. Communicating processes, components and scaleable systems. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/components-6up.pdf`, May 2001.

[2] James E. White. RFC 707: A high-level framework for network-based resource sharing, December 1975.

[3] Sun Microsystems. Java remote method invocation. `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`.

[4] R. Jungclaus, G. Saake, and C. Sernadas. Using Active Objects for Query Processing. In *Object-Oriented Databases: Analysis, Design and Construction (Proc. of the 4th IFIP WG 2.6 Working Conf. DS-4, Windermere, UK, 1990)*, pages 285–304, 1991.

[5] Patrik Reali. Structuring a compiler with active objects. In *roceedings of the Joint Modular Languages Conference on Modular Programming Languages (LNCS 1897)*, pages 250–262. Springer Verlag, 2000.

[6] K. X. S. Souza and I. S. Bonatti. Using distributed active object model to implement tmn services. In M. E. Cohen and D. L. Hudson, editors, *Proceedings of the ISCA 11th International Conference*, San Francisco, California, 1996.

[7] Active Objects in Symbian OS. `http://wiki.forum.nokia.com/index.php/ Active_Objects_in_Symbian_OS`.

[8] An Introduction to Active Objects for UIQ 3-based Phones. `http://developer.sonyericsson.com/site/global/techsupport/tipstrickscode/symbian/p_active_objects_uiq3.jsp`.

[9] T.A. Busby and L.T. Chen. Developing collaborative games using active objects. `http://jdj.sys-con.com/read/35903.htm`.

[10] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Professional, 1996.

[11] The French National Institute for Research in Computer Science and Control. ProActive: A comprehensive solution for multithreaded, parallel, distributed, and concurrent computing. `http://proactive.inria.fr/release-doc/html/index.html`.

[12] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[13] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[14] D. Kafura, M. Mukherji, and D.M. Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):337–350, 1995.

[15] D. Kafura, D. Washabaugh, and J. Nelson. Garbage collection of actors. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 126–134, 1990.

[16] D. Kafura, D. Washabaugh, and J. Nelson. Progress in the garbage collection of active objects. *ACM SIGPLAN OOPS Messenger*, 2(2):59–63, 1991.

[17] I. Puaut. A distributed garbage collection of active objects. *ACM SIGPLAN Notices*, 29(10):113–128, 1994.

[18] A. Vardhan and G. Agha. Using passive object garbage collection algorithms for garbage collection of active objects. *International Symposium on Memory Management*, pages 106–113, 2002.

[19] C. Dony, C. Urtado, and S. Vauttier. Exception handling and asynchronous active object: Issues and proposal. *Lecture Notes in Computer Science. Advanced Topics in Exception Handling Techniques*, 4119:81–100, 2006.

[20] O. Nierstrasz. Regular types for active objects. *SIGPLAN: Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–15, 1993.

[21] O. Nierstrasz and M. Papathomas. Towards a type theory for active objects. In *SIGPLAN: Proceedings of the workshop on Object-based concurrent programming at the Conference on Object Oriented Programming Systems Languages and Applications*, pages 89–93, Ottawa, Canada, 1991.

[22] F. Puntigam and C. Peter. Types for active objects with static deadlock prevention. *Fundamenta Informaticae*, 48(4):315–341, December 2001.

[23] Z. Guessoum and J. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76, 1999.

[24] O. Nierstrasz and M. Papathomas. Viewing objects as patterns of communicating agents. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications European Conference on Object-Oriented Programming (OOPSLA) (ECOOP)*, pages 38–43, Ottawa, ON CDN, [10] 1990. ACM Press , New York, NY , USA.

[25] J. Briot, R. Guerraoui, and K. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[26] B. Pryor. Implementing active objects with java dynamic proxies. `http://benpryor.com/blog/2006/03/08/implementing-active-objects-with-java-dynamic-proxies/`.

[27] C. Petitpierre. Synchronous active objects introduce CSP's primitives in Java. In James Pascoe, Peter H. Welch, Roger Loader, and Vaidy Sunderam, editors, *Proceedings of Communicating Process Architecture 2002*, pages 109–122, Reading, United Kingdom, September 2002. IOS Press.

[28] W. Chen, Z. Ying, and Z. Defu. An efficient method for expressing active object in c++. *SIGSOFT Software Engineering Notes*, 25(3):32–35, 2000.

[29] J. Dongarra. MPI: A message passing interface standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.

[30] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practise and Experience*, 12(11), September 2000.

[31] Peter H. Welch and Jeremy M. R. Martin. A csp model for java multithreading. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 114. IEEE Computer Society, 2000.

[32] Peter H. Welch. Communicating Sequential Processes for Java (JCSP) `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[33] Peter H. Welch. Process oriented design for Java: Concurrency for all. In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, volume 1, pages 51–57. CSREA, 2000.

[34] Peter H. Welch, Jo R. Aldous, and Jon Foster. Csp networking for java (jcsp.net). In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 695–708. Springer-Verlag, 2002.

[35] G. Rimassa F. Bergenti, A. Poggi and M. Somacher. DisC++: A software library for object oriented

concurrent and distributed programming.

[36] D. Kafura, M. Mukherji, and G.R. Lavender. Act++ 2.0: A class library for concurrent programming in C++ using actors. *Journal of Object Oriented Programming*, pages 47–55, October 1993.

[37] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer-Verlag, 1996.

[38] N. Benton, L. Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[39] J. Briot. Actalk : A framework for object oriented concurrent programming - design and experience. In *Proceedings of the 2nd France-Japan Workshop (OBPDC'97). Herms Science*, 1999.

[40] O.M. Nierstrasz. Active objects in hybrid. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22(12), pages 243–253, New York, NY, 1987. ACM Press.

[41] F. Matthijs, W. Joosen, J. Van Oeyen, B. Robben, and P. Verbaeten. Mobile active objects in Correlate. In *ECOOP'95 Workshop on Mobility and Replication*, Aarhus, Denmark, 1995.

[42] J. S. Dong and B. Mahony. Active objects in tcoz. In *ICFEM '98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, page 16, Washington, DC, USA, 1998. IEEE Computer Society.

[43] mpiJava. `http://www.hpjava.org/mpiJava.html`.

[44] T. Ylonen. RFC 4254: The secure shell (SSH) connection protocol, January 2006.

[45] Sun Microsystems. Java runtime environment.
`http://java.sun.com/j2se/desktopjava/ jre/index.jsp`.

[46] L. Lamport. Time, clocks and the orderings of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[47] Active Objects Tutorial.
`http://research.microsoft.com/COmega/doc/comega_tutorial_active_objects.htm`.

[48] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i: Matrix multiplication. In *Parallel Computing*, pages 17–31, 1987.