An Implementation of Active Objects in Java

By
George Oprean
Matt Pedersen

Outline

- Introduction
- Active Objects
- Related Work
- Asynchronous Active Objects in Java
- Implementation
- Results
- Conclusions
- Future Work

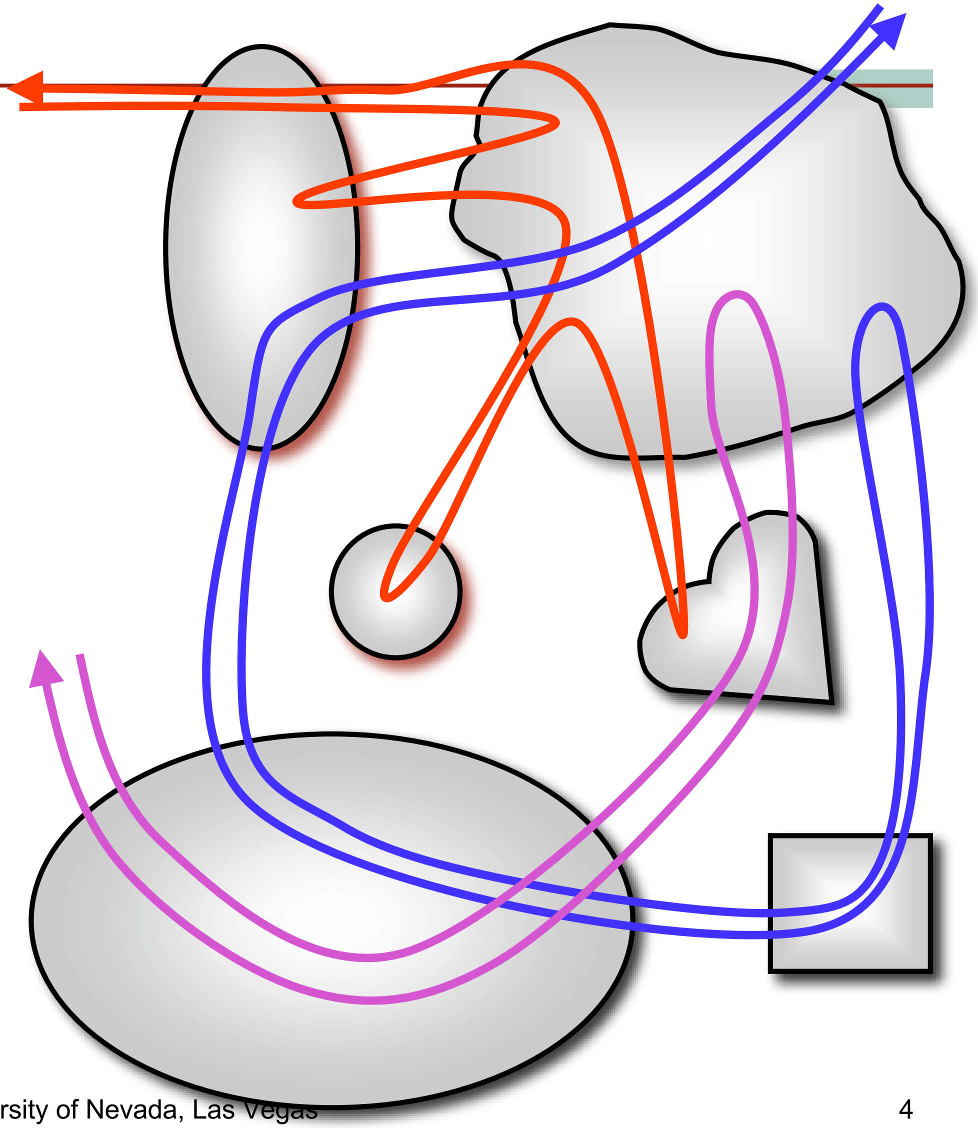
Introduction

- Object Oriented paradigm
  - widely used in the last two decades
  - models how objects interact in the real world
- objects are passive
  - friend.borrowMoney(20);
  - would reach into friends pocket and get the money
- methods are executed synchronously
  - wait until friend gives me the $20
- more than one thread can have a reference to an object, thus the object can be put in an inconsistent state

Objects Considered Harmful

Each single thread of control snakes around objects in the system, bringing them to life *transiently* as their methods are executed.

Threads cut across object boundaries leaving spaghetti-like trails, *paying no regard to the underlying structure*.



Active Objects

- executes method invocations in its own thread
- receives the message, perform the computation and return the result to the caller
- queues the requests and decide what method to execute next (order of arrival, priority)
- only one method executes at one time → object can not be put in an inconsistent state

Active Object (2)

- methods can be invoked synchronously or asynchronously
- asynchronous communication → the uses the 'waiting time' for other computations
  - waiting time = the time it takes the caller to get the result back
- preparing breakfast example:
  - no cereals? Ask the active object to get the cereals
  - meantime, get the milk, set the spoons and pour orange juice
  - got back the cereals? Breakfast is served.
- waitfor statement used for getting the result of asynchronous calls

Related Work

- employing patterns
  - Active Object or Dynamic Proxy Pattern
  - active object and pattern components have to be implemented
- extending the language with new keywords
  - Java – active, accept, select and waituntil
    - only synchronous active objects
  - C++ - active, passive
    - both synchronous and asynchronous
- using external libraries (like MPI for C)
  - ProActive library for Java

Asynchronous Active Objects in Java

- implemented our system in Java
  - the language is OO
  - it has RMI built in
  - it supports reflection
  - Java compiler available as open-source
  - it is platform independent
  - autoboxing done implicitly (from JDK 1.5)

Asynchonous Active Objects in Java (2)

- an asynchronous Java active object characteristics:
  - must be active (use own thread to execute the methods)
  - can be placed on any reachable machine on the network (ssh, JRE)
  - allow both synchronous and asynchronous method invocation
  - provide a way to obtain the result of asynchronous call

New Keywords

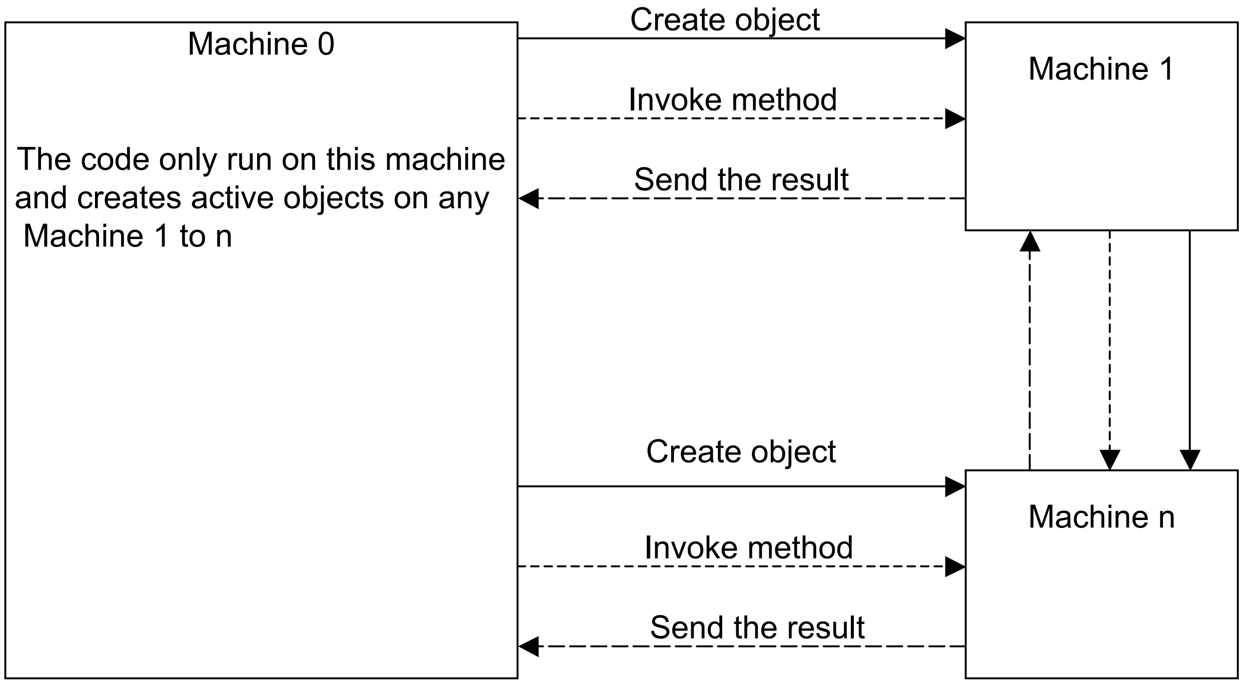
- a new **active** modifier
  - marks a class as being active
- an extended object creation
  - `actObj = new ActiveClass()` **`on`** `"machine_1";`
- an extended method invocation expression
  - `actObj.foo()` **`async;`**
- a new *blocking* **waitfor** statement
  - **`waitfor`** `actObj var;`

Restrictions on Using the New Keywords

- asynchronous invocation applies only to the last method, if method calls are chained
  - `actObj.foo().bar()` **`async;`**
- asynchronous invocations can only appear on the right side of an expression
  - illegal: `obj.method(actObj.foo()` **`async`**`)`
- waiting for the results of asynchronous invocation on the same object is the same as the order of invocation

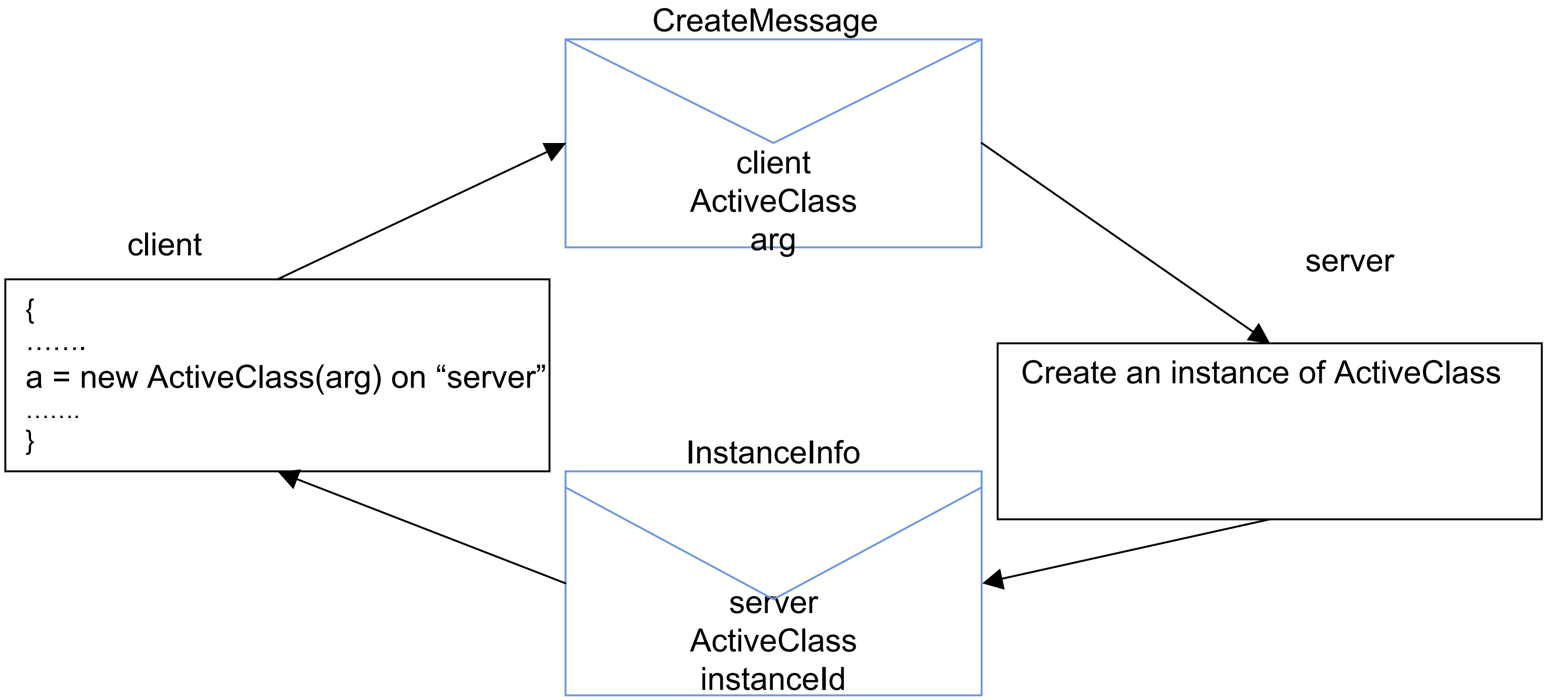
Implementation Design Overview

- communication by exchanging messages
- both synchronous and asynchronous



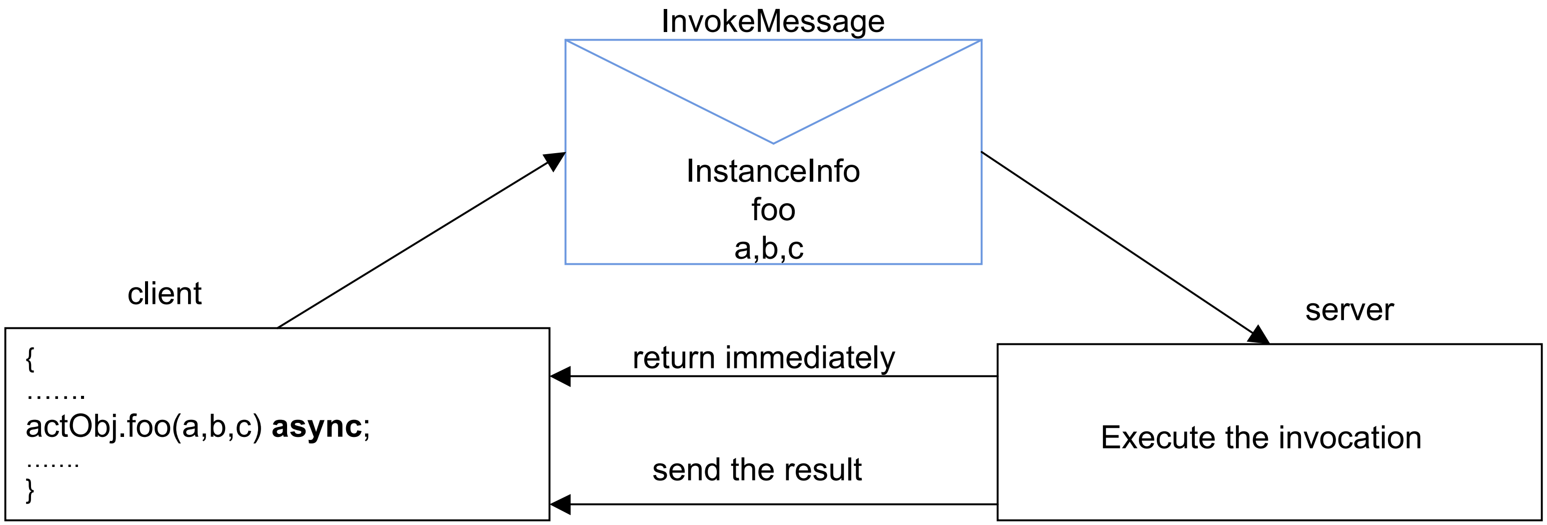
Implementation
Creating an Active Object

- `a = new ActiveClass(args) on "server";`
- synchronous communication



Implementation
Invoking Active Object's Methods

- `actObj.foo(a,b,c)` **`async;`**
- without **async** → synchronous communication



InvokeMessage

InstanceInfo
foo
a,b,c

client

```
{
…….
actObj.foo(a,b,c) async;
…….
}
```

server

return immediately

send the result

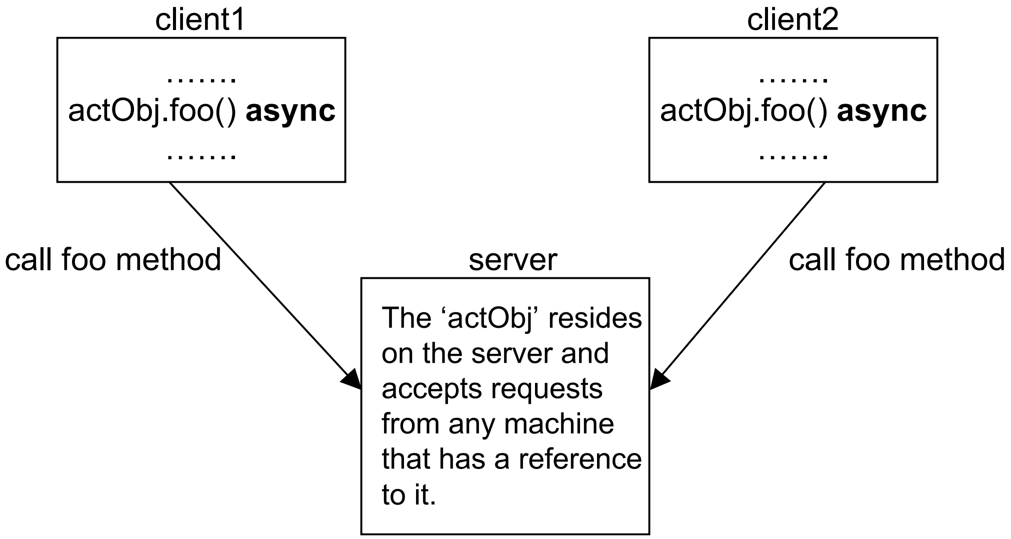
Execute the invocation

Implementation
Getting the Result of Async Calls

- waitfor actObj var;
- programmer: “I’m waiting for the result of an asynchronous invocation and I want to store the value in *var*.”
- waitfor is a blocking statement
- results of async invocations not waited for? Will be discarded when the method finishes
- wait for the result of async calls in the same method as the invocation

Implementation
Message Ordering

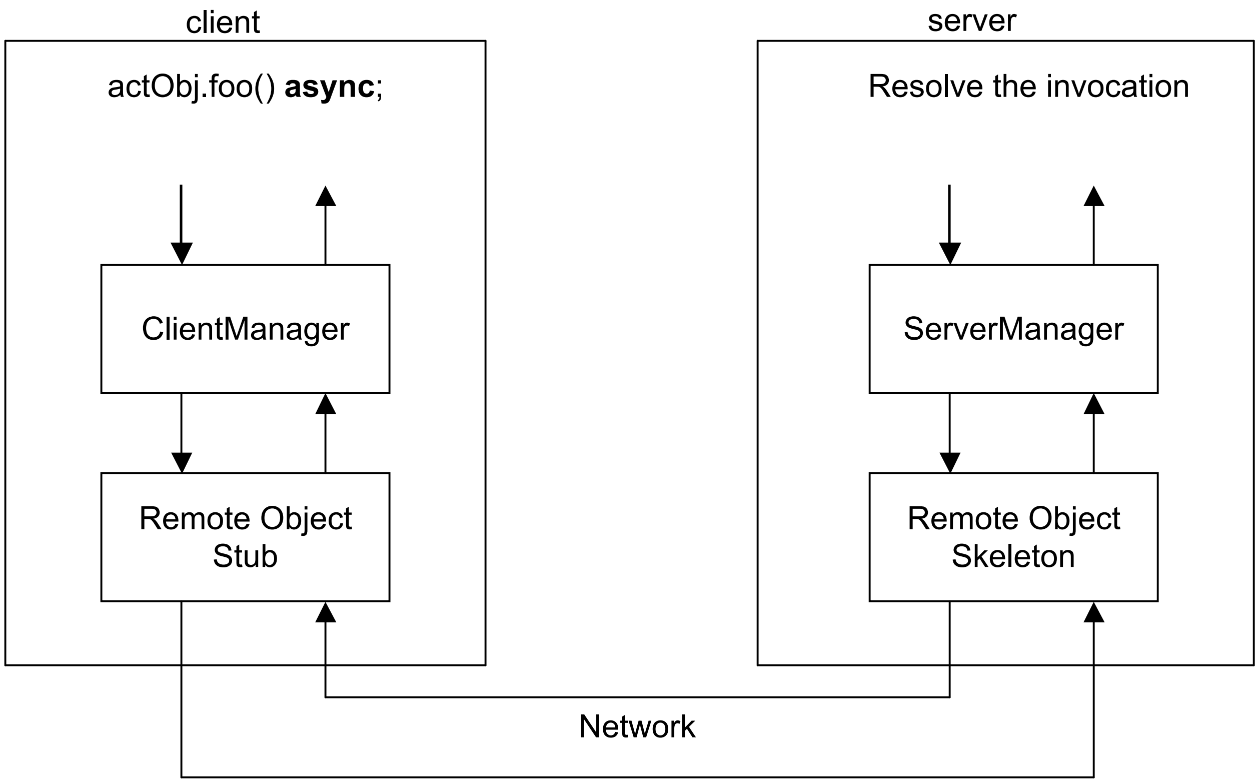
- active objects can be passed around
- only a reference is passed and not the actual object
- partial ordering: invocations from the same machine on the same object will be executed in order



The ‘actObj’ resides on the server and accepts requests from any machine that has a reference to it.

Implementation ClientManager and ServerManager

- the core components of our system

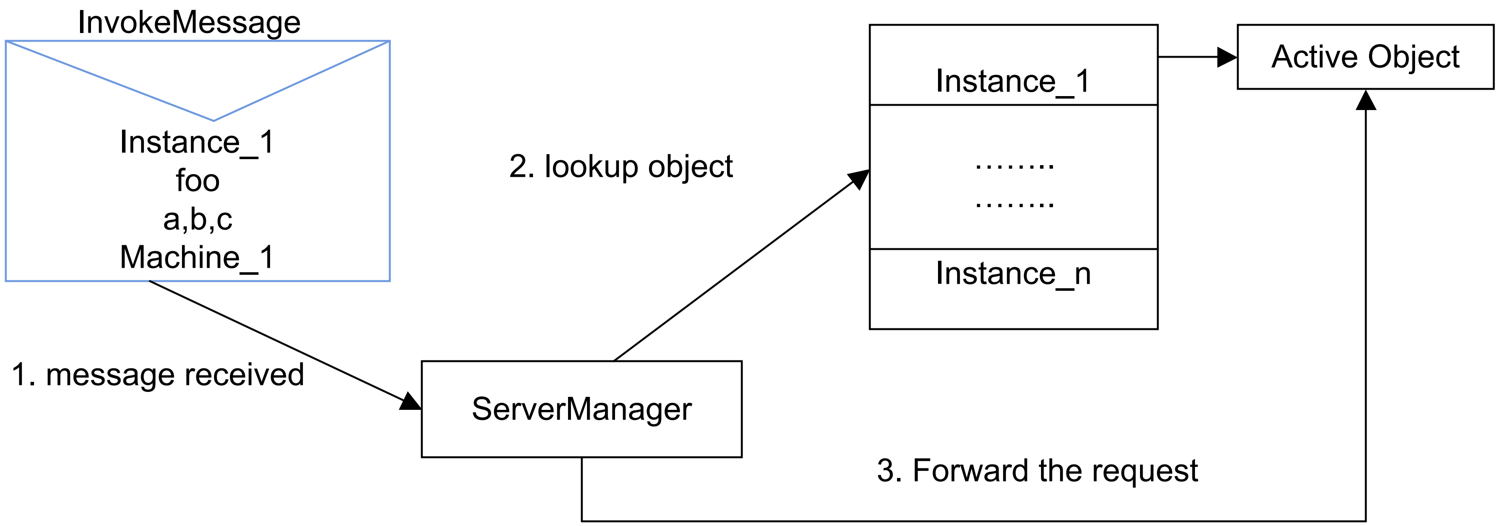


Implementation ClientManager

- only one per machine
- manages the active invocations from the machine it is running on
- manages the results of async invocations
- core functionality
  - *invokeConstructor* – creates an active object
  - *invokeMethod* – invokes a method on an active object
- additional functionality
  - *getMethodId* – each method has a unique identifier
  - *removeUnwaitedCalls* – removes unwaited results of asynchronous invocations

Implementation ServerManager

- similar role as ClientManager, but on the machine that hosts the active objects
- only one per machine
- needs to be started before the program is run (through ssh script)
- accepts create and invoke messages



Implementation
Compiler Modifications

- modified Sun’s open-source JDK 1.6 compiler
- new keywords are translated into regular Java code during desugaring phase
- **active** keyword is removed from the class definition

Implementation
Compiler Modifications (2)

- new creation expression

```
ActiveClass actObj = new ActiveClass() on “server”;
```

will be translated to

```
InstanceInfo actObj =
  ClientManager.invokeConstructor(“ActiveClass”,
                               new Object[]{}, “server”);
```

Implementation Compiler Modifications(3)

- adding the methodId declaration
  - `Long methodId = ClientManager.getMethodId();`
- modifying the async invocations:
  - `actObj.foo(a)` **`async`**`;`

  will be translated to:

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

Implementation
Compiler Modifications (4)

- modify the **waitfor** statement
  - **waitfor** actObj var;

  will be translated to

  -
    ```
    ReturnObject r0 =
              ClientManager.waitForThread(methodId, actObj);
    ```
    ```
    var = (Integer) r0.getReturnValue();
    ```
- remove the unwaited async calls
  - `ClientManager.removeUnwaitedCalls(methodId)`

Example: Subscriber / Distributor

```
public active class Distributor {
  private ArrayList<Subscriber> subscriber();
  public void Subscribe(Subscriber s) {
    subscriber.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 argv[]) {
    Distributor d = new Distributor();
    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”);
  }
}
```

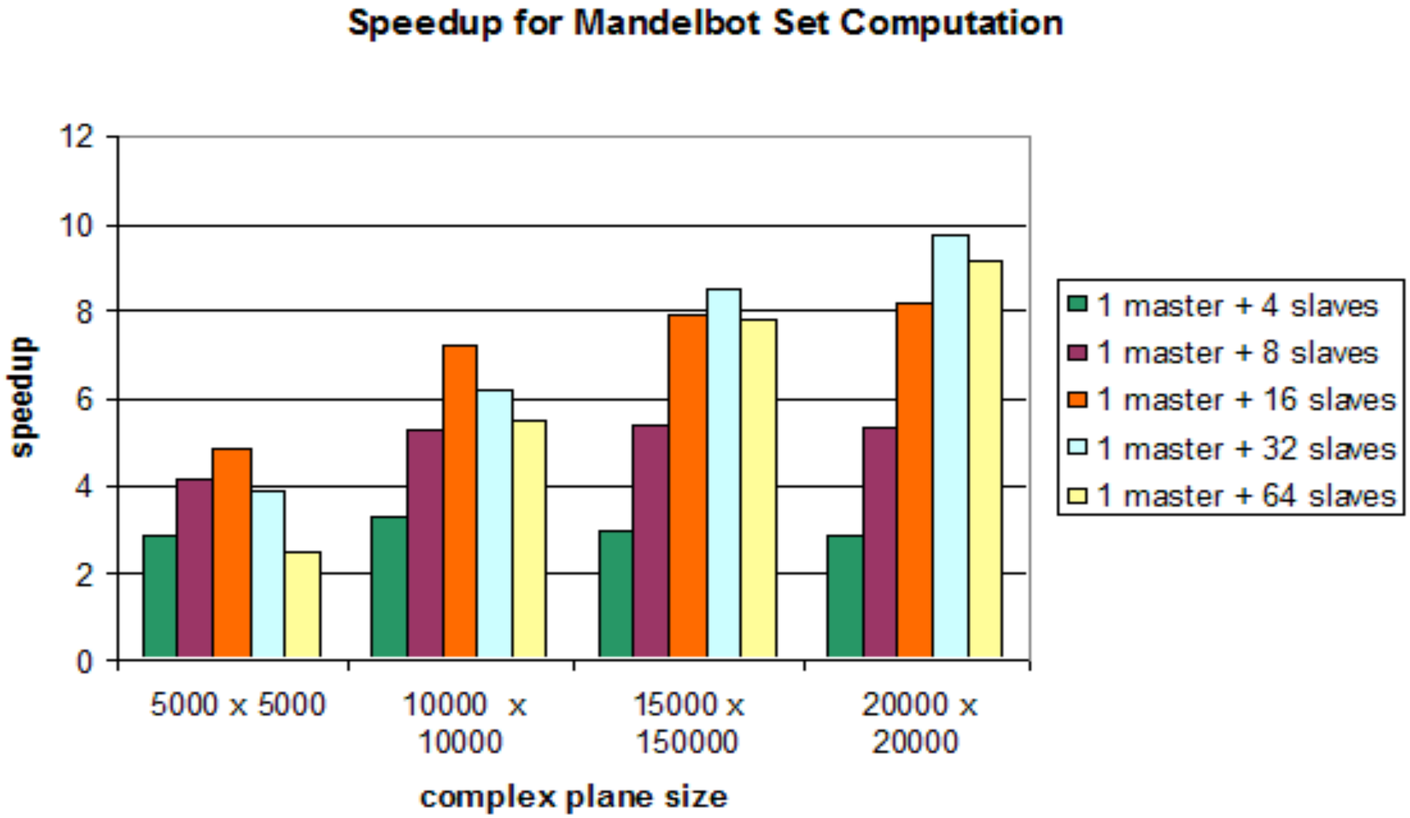
```
a got the message: First message
b got the message: Second message
b got the message: Third message
c got the message: Third message
a got the message: Second message
a got the message: Third message
```

Active Objects for Distributed Computing

- active objects used for developing parallel and distributed applications
- async invocations → parallel computation
- create objects on any machine on the network → distributed computing
- implemented Mandelbrot set computation, Matrix multiplication and Pipeline computation

Results
Mandelbrot Set Computation

- speedup= sequential time / parallel time



Conclusions

- Object Oriented programming increased popularity compared to logical or procedural programming
- objects are passive
- active objects better reflection of the world (both passive and active objects)
- extended the Java language: **active** , **async**, **on** and **waitfor**
- develop parallel and distributed applications
- results demonstrate the feasibility of our proof of concept

Future work

- our system can be extended
  - starting/stopping the ServerManager from code
  - warning the user if asynchronous calls with a return value do not have a matching waitfor
  - including an exception mechanism
  - receiving out of order invocations
  - keep active objects after the application finished the execution

Thank You!