AJ: A SYSTEM FOR BUILDING ACTORS WITH JAVA

BY

WILLIAM ROGER ZWICKY

B.S., University of Wisconsin – Platteville, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Adviser:

Professor Gul Agha

Table of Contents

Preface

When this project was started, Java was still fairly new, and using reflection to assist remote communication was a relatively unique idea. Today, these concepts are in heavy use everywhere. Java now has several actor platforms available for it, and actor libraries have been created for many other languages as well (see Chapter 5).

Nonetheless, AJ still has some value in demonstrating the shape of a highly modularized, purely Java actor environment. The modules do not impose any significant speed drop, but allow one to experiment with alternative low-level mechanisms. This report also surveys recent research into actor platforms and actor-like distributed systems.

1. Introduction

The **actor model** [1] was developed to support reasoning about concurrency. However, the actor model is so straightforward that it makes a perfectly reasonable environment for writing distributed programs. Such an "actor language" is very flexible and powerful, allowing for the creation of programs whose modules are as tightly or as loosely coupled as necessary. The model has been used to analyze advanced concepts like location-independent messaging and moving live programs between host computers, so it's only natural that an actor-based programming language implement such features.

This thesis describes AJ, a software system for writing distributed programs in Java, and is based on the actor model. In AJ, an actor is an extension of an object: where objects communicate by calling each other's methods, actors communicate by sending asynchronous messages to each other. AJ provides the messaging layers that allow actors to communicate with each other, no matter if the actors are on the same computer, or scattered across a network.

AJ provides the following features:

- Each actor is an instance of a single Java class for easy coding.
- Each actor has its own thread so that it can execute concurrently with other actors, as well as communicate asynchronously.
- Actors may be created and destroyed dynamically, just as easily as normal Java objects.
- Actors may be moved between hosts to improve load sharing or data locality.

- The message system is layered, allowing protocols and features to be plugged in. The basic system includes several features:
    - Actor *addresses* provide direct communication with an actor's host machine, for fast messaging. Actor *names* provide location-independence, allowing actors to move around a network.
    - Shared-memory messaging communicates very quickly, while Internet messaging communicates with other machines across a network.
- The message carrier class can be customized to speed processing and reduce overhead.
- The AJ run-time is not a Java API, but instead appears as a collection of actors. For example, a new actor can be created by sending a "create" message to the server actor.

```
void doWork() {
       getInput()
       process()
       updateDatabase()
       processMore()
       writeReport()
}
```

Figure 1-1: Plain Java - a single list of instructions.

Java is a natural choice for AJ due to its popularity, as well as its wide array of features. It has a powerful network library, almost bug-proof memory management, and built-in threading. It also allows high-level manipulation of the classes and objects making up a running program, which makes messaging and migration possible. Although Java object serialization is not very fast, it's sufficient for experimental systems like AJ. For more on the fundamentals of supporting actors in Java, see [31].

1.1. What's an Actor in Theory?

The actor model was developed as a theoretical foundation for reasoning about concurrency. Any parallel program, no matter what language it is written in or which processor it runs on, can be represented as a set of actors. For that matter, any sequential program can be represented with actors, but there are easier ways to analyze such programs.

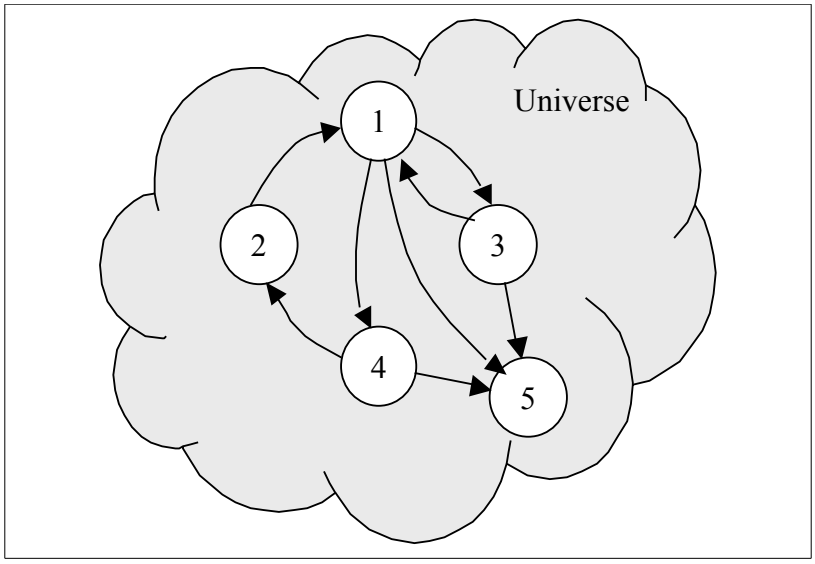

Figure 1-2: Actor program – a loose collection of independently running objects.

The heart of the actor model is the *actor*, which is an autonomous computing agent. One actor alone has no internal concurrency, but multiple actors can run in parallel. Actors communicate by sending messages to each other. Messages are not ordered or prioritized, but are guaranteed to arrive eventually. Specifically, actors have the following properties:

- Individual actors are defined by a *behavior*, which is essentially the code that the actor runs to process messages.
- Each actor executes serially. That is, an actor can have no internal concurrency.
- Actors can communicate only by sending messages to each other; hidden channels of communication (like shared memory) are not allowed.

- Messaging is asynchronous; actors don't need to wait for the message to be received or processed before continuing.

- Message delivery is fair; a message is guaranteed to arrive eventually. However, there are no other guarantees like timing or ordering.

- The medium that messages travel through does not impose any restrictions on the messages or the actors. The medium behaves like a queue for each actor, but the medium can never fill up nor lose messages.

Early work motivating actor theory includes analyzing the advantage of actors for performance in parallel and distributed systems [2][3] and applications to fault-tolerance [26], multimedia [21] and real-time systems [19]. The language Rosette [28] introduced interoperability between concurrently executing databases, scientific computing, graphics packages, and other systems.

1.2. What's an Actor in Practice?

In AJ, an actor is an instance of an extended Java class. Each actor has its own message queue, and the class definition contains the code that actually processes the incoming messages. More formally, an actor is implemented with three objects:

- **An Actor class** defines what types of messages can be accepted, and what to do with each type. This code is like the *behavior* mentioned above.

- **A Mailbox** queues incoming messages.

- **A Thread** which allows a class to run concurrently with other classes.

The actor class corresponds to the "behavior" from actor theory: it defines what to do with each message. Messages are designed to look like function calls, so that the system can automatically match a message to the Java method that handles it. The system actor defines functions for sending messages and creating actors, and even one to change the class associated with a particular actor.

The Mailbox contains a message queue that allows senders to continue processing after sending a message. Actor theory dictates that the message queue be able to hold an unlimited number of messages. This is impossible in Java of course, but AJ places no arbitrary restriction on the number of messages that can be held.

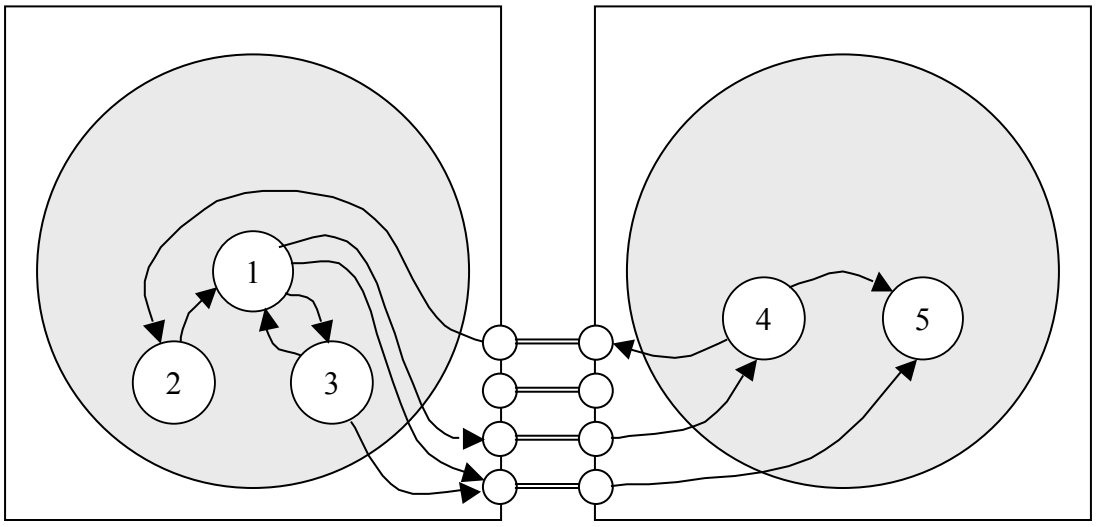

Figure 1-3: AJ program – a loose collection of objects, connected by a network.

Each actor runs in its own thread of execution, managed by an actual Thread object. The Thread object is the actors handle to the host computer's processor, and allows the actor to be started, stopped, or assigned priority to allow other actors to have more or less processor time.

1.3. Design Goals

AJ was designed for experimenting with actors, not for simply running them. Hence it was designed for clarity and modularity, rather than performance. Some of the principles used in building the code are listed below; further details are given elsewhere in this document.

- High modularity. Each of the classes in AJ is designed to be complete within its scope of functionality. This means that experimenting with AJ requires only manipulating collections of objects or plugging in new objects rather than re-writing existing classes.

- System uniformity. The AJ run-time is not a Java API, but instead appears as a collection of actors. For example, a new actor can be created by sending a "create" message to the server actor.

- Non-centralized services. In general, low-level system services only manage the node on which they run. If one node fails, others won't be affected.

- Well-defined responsibilities. The system itself is made of actors, so talking to the system only requires sending messages to the actor that handles the desired feature. Extending the system involves simply adding more actors; there's no central manager that needs to be recoded for the extensions.

- Layered communication. The most basic layer supports Message objects and moving them between actors. Higher layers take care of buffering and location independence. Any object can send messages, even if it's not an actor. However, only actors can receive messages.

- Flexible communication. Methods can be invoked through the messaging system, or via a direct Java function call. The messaging system itself is generalized to allow different protocols to be plugged in.

- Location independence. Messages can be sent to an actor by naming the actor; the exact location of the actor need not be known.

- Actor migration. While migration is not a standard part of actor theory, it's a big part of agent research. Also, Java makes migration fairly simple with built-in serialization. To support migration, an actor system will also need some way for messages to find roaming actors.

- No static members. AJ is composed of small, tightly defined classes. A static member would force all instances of a class to have common behavior, and this is not desirable. Instead, objects that need to share behavior will have an explicit member to hold a suitable object. Also, factory methods are used wherever constructors need persistent information. For example, the "system" actor is not

global; freshly created actors must be given the address of their manager before they can start. Several independent actor environments can be run on the same computer, for testing or security purposes.

- Optional overhead. Some features are pluggable, allowing the user to trade flexibility for speed. A program can be developed quickly, then refined later. Further, some of these features can be swapped on the fly, allowing a system to tune itself at run-time.

1.4. Related Work

There are a number of paradigms available for writing distributed software. Many of them at least look similar to the actor paradigm, but this is not accidental: actors were developed specifically to analyze the behavior of distributed applications in general. Furthermore, parallel software is hard to write, and while most platforms make distributed systems easier to manage, few truly make it easier to develop the software itself. As a result, most platforms break down into similar sets of concepts.

Several of these platforms will be examined in detail in chapter 5. For the purposes of comparison, they will be broadly grouped by the nature of their paradigm:

1. *Actor-like systems* treat actor theory as if it were an actual programming paradigm. The constructs available closely match those in the theory. This group includes AJ itself, along with several other projects developed at the same lab.

2. *Distributed communication systems* provide mechanisms for remote programs to communicate, but may or may not include a layer implementing the actor model.

2. Using AJ

2.1. Basic Method Walkthrough

Building an actor program is much like building a normal Java program – each actor is defined by a single class with one or more methods. But instead of directly calling these methods, an actor instead sends a message that describes the method to call. The message is transmitted to the target actor, then is placed in the target actor's message queue. The actor will process messages in the order it receives them, just as fast as it is able.

Sending an asynchronous message to an actor is simple:

```
send( address, "method", parameter, ... );
```

"Send" cannot return a value to the caller. For that, we need a synchronous *call*:

```
value = (ValueClass)call( address, "method", parameter, ... );
```

Sending a message requires knowing the actor's *address*, which wraps the data describing the location of an actor, along with the means to communicate with that actor. One way to get an address is to ask an actor server at some node to create an actor:

```
address = (Address)call( serverAddress, "create", className );
```

AJ also provides a name server which can globally associate arbitrary strings with addresses:

```
address = (Address)call( nameServer, "lookup", "global-actor-name" );
```

Both of these calls are themselves actor messages – AJ's core services are presented as a set of actors. But if communicating with one actor requires contacting another actor first, how does one get the *first* address? One solution is for an actor to simply request the address of the server hosting itself or another actor:

```
myServerAddress = this.getServer();
```

or:

```
theirServerAddress = (Address)call( theirAddress, "getServer" );
```

A second solution is available through the startup sequence of an actor node. When an actor server is first started, it can be given the class name of an actor to instantiate and boot. This actor will be sent a special "startApp" message, and the message's return address will be the server:

```
public void startApp( String args[] ) {
      Address myServer = getCurrentMessage().getSource();
}
```

The final option is to directly synthesize an address object from known information. If the actor is on the Internet, then the Internet host name and port are sufficient to access the actor:

```
serviceAddress = new IPAddress( "host.school.edu", SERVICE_PORT );
```

Finally, *receiving* a message is not a distinct command. AJ converts messages into method calls, and the method name contained in the message is automatically called. So the message "getServer" above is not a special block of code somewhere – AJ automatically calls the "getServer" method built into every actor, and marshalls and returns the result in a response message. Adding message support to an actor is as simple as adding new methods to a class.

2.2. Building an Actor Program

An actor program consists of one or more actor classes, instantiated one or more times to form the actual actors. One of the actor classes must be designed a "first" actor, and must

provide a special "startApp" method. The actor program is started by requesting an ActorServer on some computer to start the first actor. The server will send the "startApp" message to the first actor to start the rest of the program.

To build an actor for use in AJ, create a class that extends **ActorBase**, and add a startApp method along with any other desired methods. ActorBase extends Actor with all necessary mailbox code so that the user classes need only contain user code. ActorBase also provides helper methods like an asynchronous *send* and a synchronous *call* that waits for a return value or an exception.

Figure 2-1 shows code for a very simple actor. The code functions as follows:

- Line 4: A class defines an actor by implementing the Actor interface. The helper class ActorBase does this, plus it provides a complete set of basic functionality, so we only need to add our custom methods.

- Line 6: When the AJ environment first starts up, it creates the first actor, then sends it the startApp message to initialize the rest of the program. In this case, we will create one additional actor, then send it a message.

- Line 11: Method startApp was triggered by a message sent from the ActorServer that created this actor. ActorBase provides the method getCurrentMessage to retrieve that message, and getSource provides the Address of the server that originally sent the message, *i.e.* the server that is hosting this actor.

- Line 14: Now that we have the Address of our ActorServer, we can ask it to create another Actor for us. Note that we're sending an actor message, not calling a Java class. "Create" messages can be sent to any ActorServer on a network; the Actor will remain with the server, while the Address is sent back to the caller. Addresses can be transmitted to an actor on any node, while moving the Actor itself requires a more complicated process known as *migration*.

- Line 15: The new actor has been created and initialized, so we can start sending it messages. "Send" sends an asynchronous message, then continues running without

waiting for a response. "Call" waits for a return value or an exception before it continues running.

```
import osl.aj.*;


public class PingExample
   extends ActorBase
{
   public void startApp( String args[] )
       {
       Address server, actor;
       Object result = null;


       server = getCurrentMessage().getSource();
       try
           {
           actor = (Address)call( server, "create", "PingExample" );
           result = call( actor, "ping", new Integer(3) );
           }
       catch( Exception ex )
           { }
       System.out.println( "Received: " + result );
       }


   public Integer ping( Integer n )
       {
       return new Integer( n.intValue() + 100 );
       }
}
```

Figure 2-1: A simple actor.

- Lines 17-18: Exceptions can be sent through the network just like data. In this case, the exceptions aren't important, so they're ignored.

- Lines 22-25: This method is called when a "ping" message is sent. It receives one number as parameter, and returns a modified number.

2.3. Running an Actor

To start AJ and have it create one actor, enter this command line:

```
java osl.aj.FActorServer osl.aj.IPFeederFactory PingExample
```

Some scripts are included with AJ; this command line:

```
ip PingExample
```

does the same thing.

This command breaks down as follows:

- **java** starts the Java interpreter, instructing it to create and run an FActorServer.
- **osl.aj.FActorServer** is the main actor server in AJ. When run from the command line, it sets up a complete actor environment, and creates and manages actors within this environment. The "F" stands for "*Feeder*," which is a set of classes that abstract lower-level messaging protocols. This allows the server to be fully protocol-independent.
- **osl.aj.IPFeederFactory** tells the server which low-level protocol to use. IPFeederFactory will create the network components used by new actors, specifically addresses and mailboxes. The "IP" prefix means the components communicate via the Internet, specifically the TCP protocol.
- **PingExample** is the class that will be instantiated to form the first actor, which will then receive the "startApp" message to start the program. This argument is optional; in any case the server will remain available to host new and migrated actors.

2.4. Writing a Custom Message Class

AJ normally depends on Java's serialization feature to move data between computers, and reflection to match messages to methods. This can be quite slow, so AJ provides a

second mechanism. A custom message class can bypass some of the problems and provide faster messaging:

- Less serialization – Reflection requires all parameters to be formal Java objects, so numeric parameters need to be wrapped in Integer or Double classes. In Java, each object requires a separate memory allocation, and walking through all the pointers can slow serialization considerably. A custom message class can bypass reflection, so numeric parameters can be primitive data types, making serialization much faster.

- Direct method call – A message object can directly invoke a method on the target object. This saves the cost of a lookup through the reflection mechanism, and speeds message processing.

- Live messages – custom message objects can even perform some processing on the target node before dispatching its method. This violates the actor paradigm as presented by AJ, however; processing should be restricted to actors.

Figure 2-2 shows a custom message class that can be used with the code in Figure 2-1. The code breaks down as follows:

```
class PingMessage
  extends MessageBase
{
  int _val;


  static transient Class _templ[];
  static
    {
      _templ = new Class[1];
      _templ[0] = Integer.class;
    }
  public String   getMethod()    { return "ping"; }
  public Class[]  getTemplate()  { return _templ; }


  public PingMessage( int i )
    {
      _val = i;
    }


  public Object invoke( Object target )
    throws Exception
    {
      return ((ping)target).ping(new Integer(_val));
    }
}
```

Figure 2-2: A custom message class.

- Line 2: "Message" is the interface that all messages must implement, but the class "MessageBase" makes life a little easier by defining some basic functionality.
- Lines 6-13 are required by the Message interface, though they won't be used by this example.
- Lines 15-18 define a custom constructor, and initializes the message payload.

- Lines 20-24 invoke the “ping” method on the target actor. The parameter needs to be wrapped in an Integer class because “ping” was designed to be called from AnyMessage, which does not allow primitive-type parameters.

Figure 2-3 shows how to use the new class. This replaces line 15 in Figure 2-1. Note that the built-in “call” and “send” methods automatically switch between reflection and custom messages, depending on how they’re called. This is actually accomplished by using built-in class “AnyMessage,” which encapsulates the method name, the parameters, as well as the reflection code itself. The core system doesn’t need any special-case code for either type of message.

```
result = call( actor, new PingMessage(3) );
```

Figure 2-3: Using the custom message class.

2.5. Logical Addressing

Sending a message to an actor requires having an Address object for that actor. However, if the actor moves, the address becomes invalid, and communication is no longer possible. AJ supports migration, which allows an actor to move to another host while running. The **Name** class is a globally unique identifier which allows AJ to track the locations of actors as they move around the system. Where Address represents the physical location of an actor, Name represents the logical location.

Using names is as simple as running the actor system under **NameActorServer**. This automatically assigns each actor a name, and since Name is derived from the Address class, the actor code can use these names without any modification. NameActorServer provides the modified messaging system that tracks actors as they move around so that messages always arrive at their intended destination.

To run PingExample with names, enter this command line:

```
java osl.aj.NameActorServer osl.aj.IPFeederFactory PingExample
```

or use the included script:

```
name PingExample
```

2.6. Moving Actors Between Servers

Moving an actor from one server to another is quite simple:

- Send the actor the “moveTo” message, giving it the address of the server to move to.

- The actor will serialize itself into a byte packet, then send this packet to the destination server to be restored and restarted. All member variables are carried along so that the actor can resume where it left off.

- "moveTo" is sent as a message to guarantee that the actor isn't in the middle of some computation when it is moved, as an actor can only process one message at a time. This also means that an actor won't move until all messages received before the "moveTo" have been processed.

- Under the NameActorServer, the process of migration leaves behind a ForwardActor that forwards all messages to the actor at its new home. It also prods actor servers into updating their name tables to point to the new location.

See appendix D.2 for a sample actor that can bounce between any number of servers.

3. Design of AJ

AJ is written entirely in Java to support actors that are also in Java. The system consists of a large number of small classes interfaces, which both directly reflect the architecture of the actor model, but also make the system easy to understand. The modules are grouped into layers defined by Java interfaces to foster experimentation with the low-level system..

3.1. Nature of an Actor Environment

Run-time environments are generally designed either to favor static behavior, or to favor dynamic behavior. If one favors static behavior, then it generally includes a heavy-duty compiler to optimize programs before they run. If one favors dynamic behavior, then it generally includes either a lightweight compiler or an interpreter or both, along with a large run-time library of common functions.

There are three basic ways to build an environment for running actors.

1. Run-time only: Creating actors and sending messages requires explicit calls to system functions. This is the simplest to implement, but it's very difficult for the system to optimize actor programs for a particular machine. AJ and the Actor Foundry are both examples of this.

2. Complete translator: The compiler itself understands actors, and translates and optimizes actor programs for best performance. This allows a programmer to write "pure" actor programs, consisting only of actors and messages. The translator can optimize the program for a particular machine, boiling it down to single-processor code if possible.

3. Hybrid: A pre-compiler or pre-processor translates custom keywords or constructs into a lower level language. The actual work is performed by a run-

time support library. Broadway [25] uses just such a preprocessor to generate C code. HAL [11] translates Lisp-like constructs to C; the compiled program then uses the CHARM run-time library.

For maximum flexibility, AJ was implemented strictly as a run-time library. This requires a certain discipline when writing actor programs, as the library functions must be called directly by the user's program. The advantage, however, is that AJ can easily be adjusted and extended with minimum effort, sometimes even while an actor program is running.

3.2. Why Java?

AJ builds on the environment provided by Java. Java was chosen not only for its popularity, but also for the wide array of powerful features built into the system.

1. Network communication: Java has libraries for network communication built in. These include support for TCP, UDP, and local file systems.

2. Threads: Thread management is built in, plus many system classes are designed to support simultaneous access. The **Thread** object wraps and manages an execution thread, plus every object contains a monitor to govern shared access.

3. Object serialization: Any object can be automatically marshaled and sent through a network.

Java has a few shortcomings:

- The Java standard doesn't require fair thread scheduling. In fact, some implementations contain unfair scheduling, allowing one thread to hog the processor. AJ does not address this issue, instead relying on messaging to break up processor usage.

- Serialization is extremely slow, dwarfing all other overhead. Unless major changes are made in future versions of Java, this precludes AJ from being used for any serious distributed computation.

3.3. The Actor Environment

An actor program is loosely composed of actors much as a Java program is loosely composed of objects. Actors communicate through a formal message-passing system that allows individual actor objects to be located on different processors, different computers, or different countries. AJ provides the framework that allows actors to find each other and pass messages. Live actors can even be migrated between computers to improve locality.

As an example, Figure 3-1 shows some actors (among other things) running on two computers. System 1 is running a command prompt and an actor environment, and System 2 is running a calculator plus two independent actor environments. Within the actor environments, five actors are running and sending messages to each other.

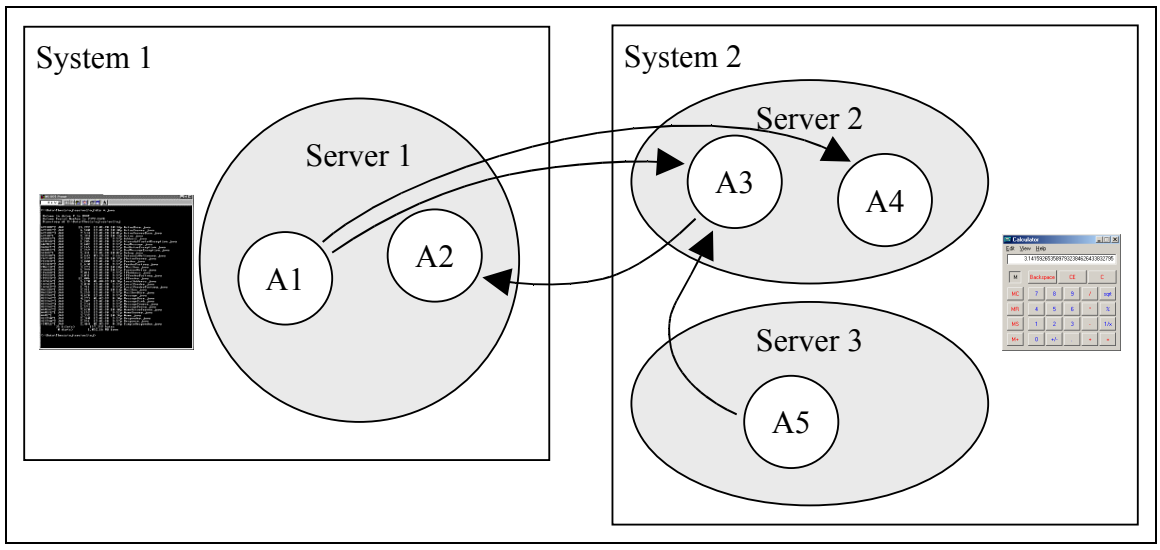

Figure 3-1: Actors running on two computers.

AJ represents each of these objects with a single high-level class:

- "System" is the computer itself, and is generally handled by the Java run-time environment.
- "Node" is represented by an **ActorServer**, which manages creation and migration of individual actors. Multiple ActorServers can run on the same system.
- Actors A1 through A5 are custom classes extended from **Actor**. The Actor interface defines the basic methods required by AJ, while the custom classes provide the application-specific code.
- The arrows represent messages, which are managed through the **Message** interface.

A complete actor program, then, consists of three parts:

- An environment for the actors to run in. This generally includes a computer, a Java run-time system, and one or more ActorServer objects. The environment provides basic services, like thread management, name services, and timers.
- A communications layer to manage the actual communication, finding actors and moving message packets between them.
- The actors themselves. Each actor is an object, an instance of some class that defines its behavior within the system.

An individual actor consists of four parts:

- A **class** defines the "behavior" of the actor, describing the messages that will be accepted, and what to do with them.
- A **thread** executes the class code when messages arrive, polling the Mailbox for new messages, and calling the appropriate methods.
- A **mailbox** queues the incoming messages, and holds them until the actor is able to process them. This allows the calling actor to continue running.

- An **address** provides the location of the Mailbox to the network layer, so that it can determine where to route messages. The Address class is actually a high-level interface, implemented as part of the communication layer. For example, an actor that lives on the Internet will have an IPAddress which defines the IP address of the hosting computer, plus a port number at that computer.

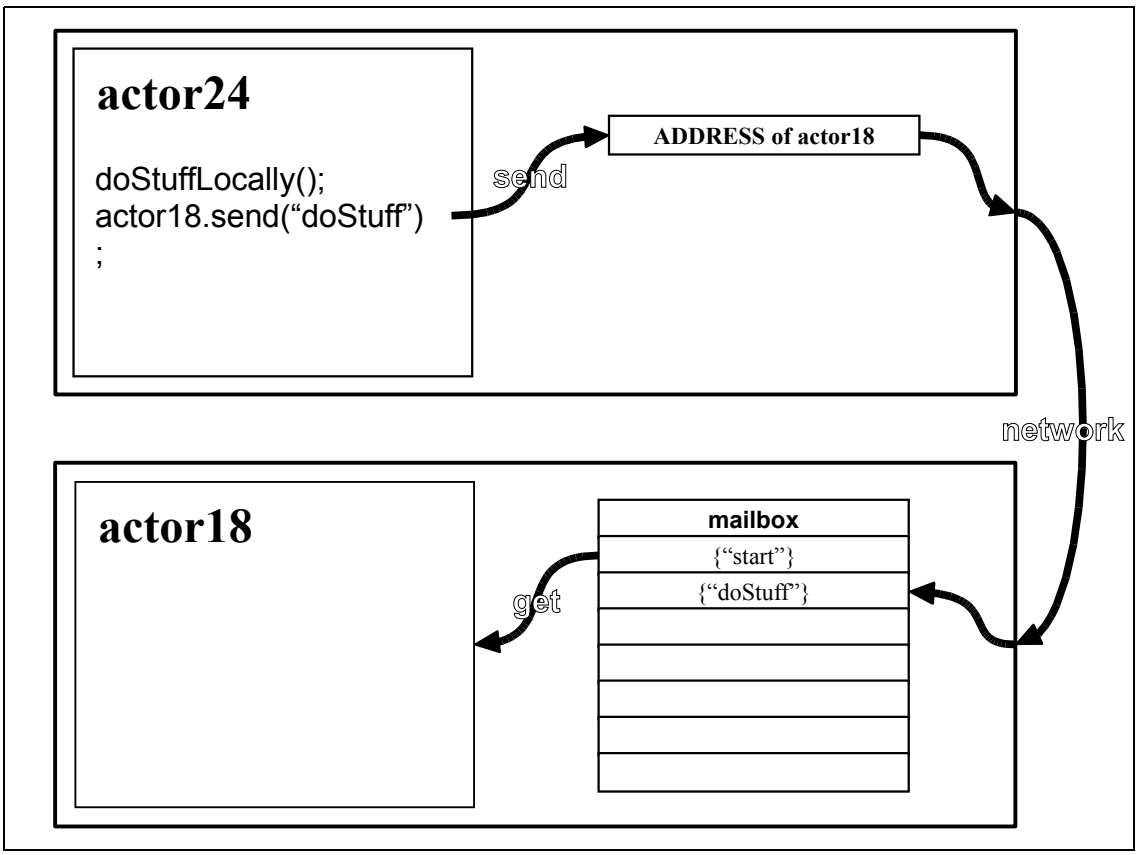

Figure 3-2: Addresses and Mailboxes form the communication layer used by Actors.

Actor communication is little more than asynchronous, buffered, remote procedure calls. A message contains the name of a remote method to call, and the parameters to pass. Synchronous messaging is possible, but AJ's real power comes from asynchronous messaging.

3.4. AJ System Architecture

AJ was designed for modularity. Most every class has an abstract interface that allows classes to be swapped in and out at will.

Starting an actor program under AJ consists of starting an actor server, telling it which communication layer to use, then instructing it to create the initial actors. Each actor's behavior is defined by a single class that accepts and processes messages. An instance of an actor class receives a mailbox and a thread to execute its methods. Each mailbox is identified by an address so that other actors can send messages to it.

3.4.1. Actors and Messaging

Each actor is managed as an independent object with its own set of resources. Actors can be created or destroyed, and added to or removed from the system at any time. Each actor has its own FIFO mailbox that queues incoming messages. There's no facility for a sender to request a priority or a position in the queue; messages are queued in the order they arrive.

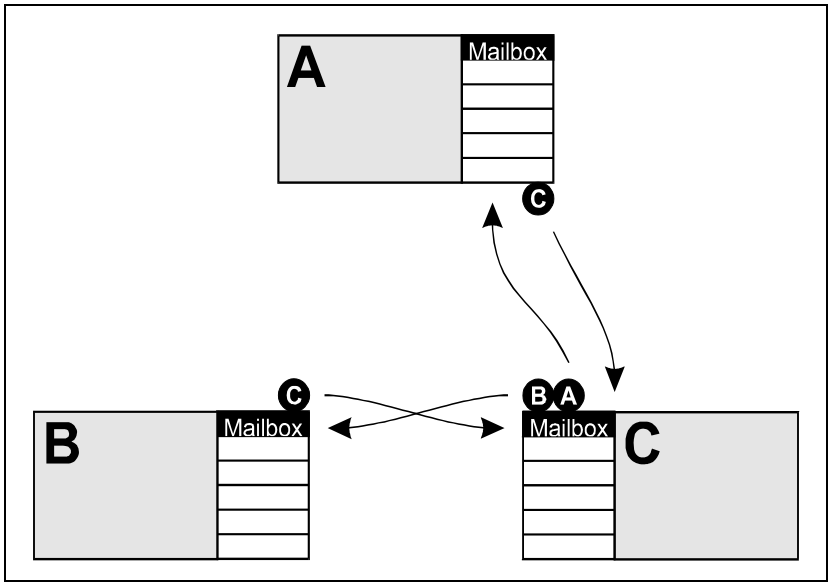

Figure 3-3: Inter-actor messaging.

Where Java objects normally communicate by directly calling each other's methods, actors communicate by passing messages; the receiver will process the message whenever it feels like, and the sender need not wait. Messages, as well as Addresses, are distinct Java objects that can be easily serialized, transmitted, and queued.

AJ supports both *send* and *call* messaging. "Call" is like a traditional Java method call – the caller stops until the called function finishes executing and returns a result. "Send" does not wait; the message is sent, then the caller resumes execution. The receiving actor will process the message when it is able. Although a sender can't receive a return value from the send, the actor that received the message may itself send a response back as a new message. Actors can trade values this way without having to stop and wait for each other. Note that true actors only communicate asynchronously since synchronous *calls* can be implemented entirely with asynchronous *sends* [1]. However, it's very convenient to have the system provide both types of messaging.

A message object contains the name of the method to call along with the complete set of parameters, while the message class contains the code for dispatching the method on an actor. This allows custom message classes to directly call a specific method, rather than going through Java reflection, for example.

Every mailbox has an associated *Address* that describes the network location of the mailbox. Address objects are created by the underlying communication layer, and encapsulate the code for sending a Message object through the network. An address doesn't actually refer to an actor, but to the mailbox that holds messages for the actor. An actor can even hand its mailbox over to another actor.

3.4.2. Actor Server

Actors are created and managed by an actor server. The server is itself an actor which represents a node that actors can run on. In general, a computer would only have one actor server, but there's nothing preventing a computer from having several, in the same or different address spaces.

To create an actor, a message needs to be sent to an actor server. Shutting down an actor requires sending a message to the actor itself.

3.4.3. Communication Modules

Actors communicate through their mailboxes; the mailboxes that come with AJ communicate through an abstract mechanism called a "feeder." When an actor system is started, it must be told which feeder classes to use. AJ provides two feeders:

- *IPFeeder* uses TCP/IP, which allows actors anywhere in the world to communicate, although it's much slower than LocalFeeder. TCP/IP is used even between actors running in the same memory space, so this effectively clones the entire message being passed. Actors can thus use the passed objects for local storage without affecting other actors.

- *LocalFeeder* uses shared memory for fast communication, although this requires that all actors be in the same memory space. This also defies the actor model, as this introduces a hidden channel of communication. This may also introduce errors, as any actor that expects a cloned message will actually be modifying a shared one.

A *Mailbox* and an *Address* are used together to form a communication link. The mailbox receives and queues incoming messages, while the address encapsulates all the code and data needed for someone else to send a message to that mailbox.

Feeders are used only for actors that use an *FMailbox* to handle messages. Although other modules can be written for the Feeder mechanism, *Address* and *Mailbox* are interfaces, and can be extended directly to maximize performance.

3.4.4. Addresses and Names

Actors identify each other by using addresses. An address contains the data needed to contact an actor along with the code to actually send a message there. Calling *Address*.**send** allows one to send anonymous messages to other actors without having to set up a mailbox.

The naming system is built on top of this, and provides a way to map string names to addresses so that actors can be found without having to know communication layer-specific information on their location. The *Name* class adheres to the *Address* interface and keeps in contact with the naming system so that actors can migrate between nodes (a process which changes the actor's address) without getting lost.

Although addresses can sometimes be synthesized with their constructors, the proper way to tell one actor about another is either for each actor to send its address in a message to whoever might need it, or for an actor to use the naming system to look up addresses it needs.

3.5. Internal Processes

AJ has many features and operations, but in many cases the operations are not coded in just one method. Instead, they're spread among several classes, requiring communication between the different layers of the system. This section describes how the layers and modules work together to implement each of the system's main features.

3.5.1. System Startup

An actor program can be started by issuing the following command:

```
java osl.aj.FActorServer osl.aj.IPFeederFactory PingExample
```

This starts the FActorServer as a program, which establishes itself as an actor environment. The first parameter to FActorServer is required, as it specifies the FeederFactory class that will build the communication layer for this server. The second parameter, if present, gives the name of an actor class to instantiate and run within the new server. Subsequent parameters are passed to that actor in a "start" message.

After initialization, the actor server itself prints its own address to the Java console. An actor generally needs an Address object before it can communicate with another actor, but if two actors are started independently on different machines, the actors can't send messages to each other until they have each other's addresses, which they won't get until

they send a message. This deadlock is broken by the user giving the printed form of the address as a parameter to the other actor environments, allowing communication to get started. IPFeederFactory also favors a default IP port for servers, making the command line unnecessary in some cases.

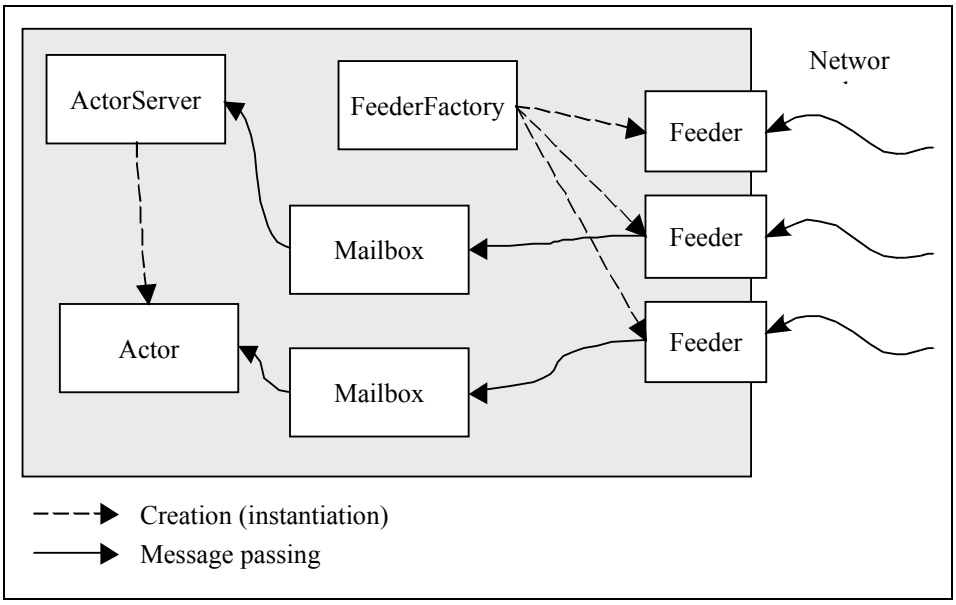

Figure 3-4: Actor server creation and initialization.

As an example, this command line:

```
java FActorServer IPFeederFactory PingExample server3.uiuc.edu 2201
```

starts a PingExample actor, telling it about an actor server running on port 2201 on a machine called "server3.uiuc.edu." Note that the syntax does not identify the nature of the communication layer required; PingExample itself must know how to synthesize a new IPAddress from the given parameters. This is not a hard requirement, however; it would be easy to build a higher-level feeder factory which takes specially formatted strings, like "ip:server3.uiuc.edu:2201", and calls the appropriate lower-level feeder factory.

When started from the command line, FActorServer performs the following tasks to create a new actor server:

- Create a new instance of FActorServer. The Java command line calls a method named "main" in FactorServer, but this is a static method; no objects are instantiated automatically. The new instance will be installed as the actor server.

- Instantiate the feeder factory class given as the first parameter. For every new actor that this server creates, this factory will be asked to create a "feeder", which connects the message-based actors to the underlying network layer, and collects incoming messages. Each Feeder class has a corresponding Address; the Address object will be sent around the network as the address of the actor itself. The Address object serializes Message objects, and sends them through the network; the Feeder object receives the byte stream and turns it back in to a Message. Address objects are serializable and clonable; they can be duplicated and sent anywhere on a network, and they will be able to send messages back to their actor.

- Create a new feeder for the server. Since an actor server is also an actor, the startup code immediately creates a feeder for itself, using a special createServer method that tries to connect to a standardized port number on the host computer. If that fails, the server creates an anonymous feeder at any available port number.

- Create a new mailbox and attach the feeder. The mailbox buffers messages that arrive from the feeder. The server can then process these messages at its leisure.

- Finally, create a process thread. The thread monitors the mailbox, and processes messages as they arrive.

3.5.2. Actor Creation

To create an actor, a server performs the same steps it did to start itself up. "Create" is a message accepted by the actor server, and can be sent by another actor from a remote location. It's even possible to instantiate the actor class on one computer (not as actor, just as an object), initialize the contents of the object, then send it to another computer to be activated as an actor.

- Create a new instance of the object, if necessary.

- Create a new feeder for the actor. The address of this feeder will be returned to the caller so that it can send messages to the new actor.
- Create a new mailbox and attach the feeder.
- Create a process thread, and activate it so it can start processing messages.
- Finally, send a "start" message to the actor to inform it that it has just been created. The return address on this message is for the actor server; this tells the actor where it's running.

3.5.3. Message Sending

In AJ, a message is sent by sending an instance of a Message object through the communication layer. Since Message is just a Java interface, a more specific class must be created first, but the implementation can be tuned to the communication layer and/or the needs of the actor program. At its most abstract, the only thing the Message object needs to do is invoke a method in an Actor object. The method and parameters can be embedded in the message object in any way, so long as the receiving end has a copy of the Java class. (Java serialization can marshal and send instances through a wire, but a copy of the class must already exist at the receiving end.)

AJ provides a default message class called AnyMessage, which uses Java reflection to invoke a named method, and pass in parameters. Java reflection works by keeping all class definitions in memory, and providing access to these definitions to user programs. Methods can be located by providing a name plus a list of data types that the desired method requires. If several methods match, Java will return the "best" match according to the same rules that the Java compiler itself uses.

One of the test programs (see appendix D.1) uses a custom class to pass messages. The class has a custom *invoke* method, which directly calls the target method on an actor. This is faster than AnyMessage, but the class can only work with the specific actor class for which it was designed. It will fail with any other class.

The transmitting side follows this procedure:

- Create an AnyMessage; specify name of target method and parameters method will require. Parameters must be formal objects; the reflection system itself prohibits primitive types like numbers and characters.
- Call address.send(message) or address.call(message) using the address of the intended recipient. Either method will:
  - Serialize the message object as a stream of bytes use in built-in Java serialization
  - Send the bytes through the communication layer.
  - The synchronous *call* method waits for a special Response message, which wraps the result of the method invoked on the target actor, whether return value or exception. Mailbox is specially designed to intercept Response objects and bypasses the regular message queue, as the calling actor thread is blocked until this response arrives.
  - The asynchronous *send* method allows the caller to continue processing, but a result will never be returned from the invoked method.

On the receiving side:

- The communication layer receives the byte stream, and de-serializes it back into an object, and passes it to the mail queue of the intended recipient. For efficiency, an actor's thread is blocked when its queue is empty. The arrival of a message unblocks the thread, allowing the actor to run until all messages are processed.
- The actor fetches the next message from the queue, and calls the *invoke* method on it, passing itself as a parameter.

- *Invoke* uses Java reflection to find the requested method in the actor's class definition. If a matching method is found, it's called with the parameters that were included in the AnyMessage object.

- If the original call was synchronous, the *invoke* method wraps any return value or exception in a Response object, and sends it back to the calling actor. If the call was an asynchronous *send*, then *invoke* quietly exits.

- The message object is disposed, and the actor proceeds to the next message in the queue.

3.5.4. Naming System

The name server merely adds a layer to the communication system; the actors otherwise continue functioning the same. The NameServer class maps logical name strings onto physical Address objects, and is updated whenever an actor moves. A Name object implements the Address interface, allowing actors to use it without requiring special code. Any attempt to send a message through a Name, however, may require a call to the NameServer to find the actor's network location. To help speed things up, a Name also contains a copy of the actors last known Address; a name lookup is only necessary if that Address stops responding.

The naming system ties itself into existing hooks in AJ, so using it requires minimal changes to an actor program's code, and no changes to the system classes:

- Setting up the actor environment:

    - Create a NameActorServer instead of FactorServer. The NameActorServer creates its own NameServer on the same node.

- Creating an actor:

    - Ask NameActorServer to create an Actor using the standard create() message. A unique name string will be generated, and placed in server's local

NameServer. The method will return a new Name object, which adheres to the Address interface, requiring no changes to code.

- Sending a message:
  - Use the standard send() and call() methods, passing the Name object as the address of the target actor.
  - The mailbox will check the lastKnown cache in the Name object, and attempt to send the message there. If the address is invalid, the mailbox looks up the name in the local NameServer, and attempts to send the message there. If that fails, an exception is thrown.
- Destroying an actor:
  - Send the standard quit() message to the actor. Currently, AJ does not automatically clean up the name servers.

3.5.5. Actor Migration

Actor migration can cause problems for an actor program, as other actors will not know where to send messages. Thus actors leave behind a ForwardActor at the old address to forward messages, and update name servers if necessary.

- Moving the actor:
  - Send the moveTo(destServer) message to an actor, where "destServer" is an Address for an actor server on another node.
  - The actor sends itself inside a create() message to the given destServer. The old mailbox is left behind.
  - The new actor server de-serializes the package, and creates a new actor with a new mailbox. A new physical address is constructed, and sent back to the calling actor. The local name server is updated with the actor's new address.

- The old actor builds a new ForwardActor, and hands over its current mailbox. The ForwardActor immediately starts bouncing messages to the actor's new physical location.
- The old actor terminates itself.

- Sending a message to a relocated actor:
  - If any actor attempts to send a message to an actor's old physical address, the ForwardActor intercepts the message, and forwards it to the actors new address.
  - The ForwardActor creates a new ForwardUpdator actor, which locates the calling actor's name server, and updates it with the new physical address. A separate actor is used to prevent deadlock between the calling actor, its server, and the ForwardActor.

4. Analysis

4.1. Performance

All tests were performed on a personal computer with a Pentium III at 600 MHz, 192 megabytes of RAM, and running Sun Java 1.3.0 (with just-in-time compiler) under Windows ME. Three tests were run to give an idea of how AJ performs versus the native performance of Java.

Test	Feeder	Custom Message	Iterations[1]	Iterations per second
1	IP	No	200,000	540
2	Local	No	200,000	2,700
3	IP	Yes	200,000	980
4	Local	Yes	200,000	12,500
5	Direct Call	n/a	10,000,000	640,000

All five tests are functionally identical. These tests show the cost of running an actor program, and where the costs lie. The program itself is a modified version of the actor listed in Appendix A in which one actor sends a message to another and waits for a response, but nothing is printed to the screen. One iteration is actually two messages – the request Message, and the Response containing a return value. The first 4 iterations of each test are not timed to give the Java run-time a chance to load in all necessary classes.

- **Test 1** is a formal, network-capability test. Messages are sent between TCP/IP ports using the general-purpose AnyMessage container. Both actors are running on the same server on one computer to eliminate the overhead of the network wire itself.

1 Iterations vary to yield a reasonable running time for each test.

- **Test 2** is the same as test 1, but the communication layer is replaced with a "LocalFeederFactory" which uses shared pointers (without cloning) to send messages, and saves the entire cost of serialization. This is not legal according to the actor model, as there shouldn't be any hidden channels of communication between actors (i.e. by sharing pointers), but it shows how much Java serialization costs.

- **Test 3** is also the same as test 1, except that it uses a custom class for passing messages. AnyMessage uses Java reflection to look up the method to call; the custom class directly calls the target method. Returned values still use the generic mechanism. The custom class also reduces the amount of serialization that must be performed.

- **Test 4** is a combination of both tests 2 and 3. This is the fastest possible speed this actor program can run.

- **Test 5** runs a fake actor that doesn't use AJ at all, but instead directly calls the *ping* method on the target actor. This test shows what plain Java is capable of, and what AJ costs.

4.2. Efficiency

Upon profiling AJ, it became clear that there were several bottlenecks and bugs.

1. **Java serialization and networking.** Network traffic is naturally slow, but Java serialization and deserialization imposes an additional performance hit.

2. **Java reflection.** Most of the time was spent fetching a complete list of method names and parameter types; it only took an additional 5% time to determine which method to call.

3. **Memory leaks.** ObjectOutputStream caches every object it touches, so that if any object appears a second time in the stream, a suitable reference token can be written. Since AJ never closes its streams, this causes any long-running program

to run out of memory. The class supports a reset() method that clears the cache and releases the memory, but this causes the system to slow down by roughly a factor of 7.

4. **Socket leaks.** Every Address created consumes another socket. If an Address instance is sent though a network stream, a new instance will be deserialized on the other end, consuming far more objects, threads, and sockets than strictly necessary. There are only a finite number of sockets available, plus some operating systems place an artificial limit on sockets[2]. This guarantees that at some point, every AJ application will eventually crash. This can be avoided by closing the socket after every use, but this causes the system to slow down by a factor of 8.

5. **Unfair threads.** As described in the documentation for the Actor Foundry [5], Java implementations are not required to have fair threading models. AJ does not address this problem at all.

This implies a few big things users can do to improve performance.

- Use LocalFeeder instead of the remote-capable IPFeeder. Unfortunately, LocalFeeder breaks from standard actor behavior by failing to clone the messages it delivers: the recipient receives references to the sender's data, creating a hidden channel of communication. Modifying LocalFeeder to clone objects makes it little faster than IPFeeder.

- Use custom message classes. Not only does a custom *invoke* save the overhead of Java reflection, but serialization can be accelerated by using primitive members to carry parameter data. AnyMessage requires each parameter to be a separate Object.

---

2 Some versions of Microsoft Windows can only open ports between 1024 and 5000. AJ will consume all of these, then crash. See: http://technet.microsoft.com/en-us/library/bb726981.aspx

- Stick to coarse-grain parallelism between physical computers. Serialization is so expensive that it's best amortized over a large amount of computation.

AJ as a framework needs a new communication layer to replace the IPFeeder, which is far too slow and resource hungry for real use. It also need a thread manager to ensure threads are run fairly.

4.3. Stability

Software stability involves more than just making sure a program doesn't crash. There are three additional issues that were considered while AJ was being developed.

4.3.1. Exceptions

AJ extends Java exception handling by allowing exceptions to be passed back to a remote caller. If a message results in an exception in an actor, and the message was synchronous (the sender is waiting for a response,) then the exception itself is returned so that it gets thrown to the sender. If the message was asynchronous, there's no way to return the exception, so it's simply printed to the console attached to the offending actor, and the system continues.

4.3.2. Lost Messages

AJ has no built-in support for coping with lost messages, as it depends on TCP/IP to deliver the data. TCP guarantees packet delivery, so no other support is needed. If other network layers are used, special attention will need to be given to network errors.

4.3.3. Deadlock

Developing a program with actors reduces the opportunity for deadlock to occur, but doesn't eliminate it. If a cycle of actors send messages to each other and wait for a response, none of the actors will be able to continue. AJ does not contain built-in support for deadlock detection. However, it would be possible to design a separate actor which is capable of breaking deadlock should it occur between specific actors.

5. Related Work

5.1. Actor-Like Systems

The actor paradigm was developed to analyze distributed computing, regardless of the computer architecture or programming language used. The core concepts, however, are fairly concrete, and can be assembled into an actual software framework. Unsurprisingly, most of these frameworks have been developed at the OSL, the home of the actor paradigm itself. The subject of this thesis (AJ) was among the first to be developed at the OSL, but there are several others.

5.1.1. The Actor Foundry

The Actor Foundry [5] is an actor system for Java that was developed at the OSL research lab, just like AJ. The Foundry was developed to support research into meta-actors, but the core itself is very similar to AJ. The concept of *meta-actors* allows policies to be coded as actors that have control over the basic operations in an actor system. These policies can cover fault-tolerance, resource management, and other system-wide concerns. As a system, the Foundry is a more polished platform, avoiding many of the problems AJ has (socket and memory consumption) and solving a few AJ never addressed (fair threading.)

5.1.1.1. Architecture

At a high level, the Actor Foundry is very similar to AJ. Both define an actor as a behavior class, a thread, and a mailbox. Both support synchronous and asynchronous message passing. Both break the run-time environment down into similar sets of modules. But under the hood, the implementation details vary greatly.

AJ's motto is “as distributed as possible,” since most of the key modules are spread across objects attached to each and every actor. The objects are created by factory

classes, making it relatively easy to experiment with different subsystems, though the selection and configuration of the factories must be done in code. The Actor Foundry takes a more centralized approach, with each module governed by a single object. The exact selection of modules that compose a foundry node is given by a text file, making it easy to experiment with different run-time environments.

The Actor Foundry run-time consists of classes from seven categories:

- The **Actor Manager**. As its name indicates, the *actor manager* manages all actors on a node. Generally each computer in a cluster will host a single actor manager, though multiple managers per node is possible. AJ calls this ActorServer, but the Foundry's manager is a much more active device: all messages must pass through the manager. While this might appear to be a bottleneck, this more accurately matches the physical hardware, as most physical systems have only a single network link. AJ's distributed Address and Feeder objects create large numbers of unnecessary threads that need to fight for that single physical link.

- The **Actor Implementation** classes. Both the Foundry and AJ have a parent class that all actor classes must descend from. The Foundry calls this "Actor" while AJ calls it "ActorBase," but both parent classes provide essentially the same set of services: actor lifecycle, access to the node manager, and message exchange.

- **Service** modules. The Foundry can be extended by plugging in additional service classes. These classes are not actors, and are only available to actors running on that node. AJ has no literal equivalent, though services can be written as actors with well-known names or addresses.

- The **Request Handler**. In the Foundry, all actors must run their messages through the manager to get delivered. The manager in turns delegates the actual process of moving the bytes from one node to another to the *request handler*. AJ uses a notably different organization (with Address as the main entry point,) but in the end both platforms provide "call" and "send" methods with the same semantics.

- The **Name Service**. This service maps symbolic names to physical addresses. This is an integral part of the Foundry, while AJ treats it as an optional communication layer. It's purpose is the same on both platforms, though.

- The **Transport Layer**. This is the low-level communications layer. While the exact implementation can be specified in the config file, this layer is never directly accessed by the actors. In AJ, this level is abstracted by the Feeder system, though it's more accessible and visible in AJ than in the Foundry.

- The **Scheduler**. The Foundry provides an explicit scheduling layer not only for research purposes, but to work around limitations in Java's own thread scheduler. AJ has no equivalent; it simply relies on Java threads.

5.1.1.2. Performance

Message passing is the biggest consumer of time in the Actor Foundry as well as AJ. While AJ uses some very simple mechanisms for communication, the Foundry uses a more carefully designed approach that avoids AJ's problems while providing a much higher throughput. Nonetheless, the results hold a surprise:

Test	Iterations	Widey Iter/sec	Littleguy Iter/sec
AJ Pinger	10,000	4,417	1,286
AJ Pinger + custom message	10,000	5,715	1,895
AF Pinger	10,000	8,113	1,126
AF Pinger	200,000	10,428	1,454

“Widey” is a laptop with a 2.1 GHz Pentium M processor, while “Littleguy” is a laptop with a 600 MHz Pentium 3 processor. These benchmarks show Widey is faster than Littleguy, as would be expected. While the Actor Foundry should be faster than AJ in theory, this turns out to only be true on Widey. On Littleguy, AJ is able to beat out the Foundry. The explanation for this is not obvious, and would likely require a detailed profiling of both platforms.

5.1.2. SALSA

SALSA [29][30] was also developed at the OSL lab to research the concept of the "World-Wide Computer." While the system is effectively equivalent to AJ and the Actor Foundry, SALSA examines some practical features in greater depth, and provides a more complete environment for complete actor applications.

SALSA focuses on four concepts: the SALSA language, continuations, messengers, and the "world-wide computer." The SALSA language is Java plus a few extensions which provide a dedicated syntax for the management and communication of actors. The communication syntax is especially key, as it provides synchronous, asynchronous, and chained messaging with a minimum of language constructs. While AJ makes messaging simple, it nonetheless requires the user to manipulate the runtime API by hand. Furthermore, AJ provides no support for chained messages.

SALSA provides direct support for continuations, which are implemented as chained messages. A simple syntax causes messages to be sent in order, with each waiting for the previous to complete, and each passing the results on to the next actor in the chain. The infrastructure is built automatically by the compiler; the user does not need to add any "chain" support to their actors.

"Messengers" are a form of active networking, introduced below with the discussion of the ANTS project. While messengers in SALSA are generally described as a carrier for messages, they are capable of performing additional processing. In ANTS, the actor is the message, and is free to perform any arbitrary processing. AJ provides no explicit support for messengers, but it supports migration, and a messenger is merely an actor that migrates to its target node before sending any messages.

The World Wide Computer is an examination of the implications of running actors on nodes distributed around the globe. Several issues are examined, including protocols, naming, and coordination, but security and failure management are left as future research.

5.1.3. Actor Architecture

Actor Architecture [12] is a simple, straightforward implementation of an actor runtime whose construction is very similar to Actor Foundry in that all messages pass through centralized managers. AA is not readily reconfigurable like AF and AJ are, but it's otherwise very similar, and provides the same set of services. AA also introduces the concept of "middle services" in the form of a directory manager which provides matchmaking and brokering services, plus a GUI to help start up and experiment with actors.

5.1.4. Erlang

AJ is the result of adding the actor framework to an object-oriented language. But actors were not designed as an implementation framework; they are a theoretical framework. A functional language would be a more natural fit for actor concepts. If the actor framework is added to a functional language, the result is Erlang.

Erlang [4][37] was originally developed for telecommunication switching systems, but along the way it picked up all of the concepts in the actor framework, along with several of the more interesting features studied by the Java-based actor run-times. In Erlang, an actor is defined as a "process" which consists of a thread, a mailbox, and a function. A function is an implementation of what actor theory calls the "behavior," thus an Erlang process is an exact match for an actor. Erlang also deals with the issues that appear when trying to make a practical actor environment: fair threading, soft real-time constraints, actor naming, actor lifecycle, remote communication, unreliable communication, behavior changing, migration, and application composition. Erlang supports continuations in the form of tail recursion, however SALSA's continuations are a bit more powerful in that the function chain can be specified separately from the functions themselves.

Erlang is missing a few features that, while not part of the actor framework, have become useful in practice: process migration and code transmission. Process migration allows a running process to move from one runtime node to another, while maintaining

communication links. Raw process migration is as simple as sending the next message to a remote process rather than a local one, however Erlang has no global address book that would allow actors to find actors that have moved. Further, running an actor on a remote system, whether by creation or migration, requires that the executable code be available to that system. Erlang cannot transmit code with a message, so all code must be in place before nodes are started.

5.1.5. ActorScript

ActorScript [9] is another OSL project to study actor features implemented on top of the JavaScript language. It introduces *ActorSpec* as a separate language which defines the composition of an entire application, and a "generative framework" that statically translates a set of ActorSpec and ActorScript prototypes into JavaScript. The project studies the implications of optimizing a web application using ActorSpec, as well as "content-aware applications."

JavaScript is halfway between Java (which formally distinguishes between classes and instances) and E (where every instance is its own class). JavaScript objects have *prototypes*, but these can be freely altered at run-time, and methods can be directly added to instances. ActorScript tightens this up slightly by introducing the "prototype" keyword, but otherwise matches JavaScript. In practice, an ActorScript actor is much the same as those defined by the other OSL projects: an actor consists of a behavior (the class or prototype containing the code), a message queue, and a CPU thread. An actor can create and destroy actors, and send synchronous and asynchronous messages. ActorScript tweaks the semantics of "class," "queue," and "thread" partly due to limitations on some of the nodes (i.e. JavaScript does not support threads) and partly to keep a lid on resource consumption (i.e. server nodes use a thread pool rather than allocating a thread to every actor.) ActorScript does not support migration, though an actor can be moved once immediately after construction.

ActorSpec is a simple language that tunes an actor-based application to a particular set of nodes, allowing adjustments for CPU speed, communication speed and latency, and

database location and speed. This allows the application code to be separated from its composition logic, allowing the same actors to be reused in different environments. Further, by building on JavaScript, ActorScript actors can be freely located in browsers, dedicated servers, or even mobile phones simply by tweaking the ActorSpec.

ActorSpec is studied in the context of web applications, where actors may be placed on the client node (where the user is located) or the server node (where the source data is located). The notion of "content-aware applications" is discussed, which are further optimized for their context (actual content, performance and capabilities of the compute nodes, etc.)

5.1.6. Compilers and Translators

The actor paradigm need not be exposed to the programmer. If exposed, the executable binary need not implement the actors literally as given in the source code. A compiler can optimize actor-like source code into a form more suitable for modern processors and clusters by, for example, converting messages to function calls and sharing mailboxes between actors. Two such projects at the OSL, HAL [11] and THAL [14], found that a good compiler can yield efficiency close to a good C implementation [15]. Other efficient actor languages include Concurrent Aggregates [10] and ABCL [27][17].

5.1.7. Other Actor Extensions

Actor support has been added to many other languages. Stage adds actors to Python [6]; Actalk adds them to Smalltalk [7][8]; Act++ adds them to C++ [13]; and ABCL adds them to Lisp.

5.2. Distributed Communication Systems

5.2.1. ANTS

In AJ, a message is an ordinary passive object that actors work with. But if a message was itself an actor, then all communication would be performed by migrating actors.

This concept is introduced as a "universe of actors" in the actor model, and as a "messenger" in SALSA.

ANTS (Active Node Transfer System) [33][34] is a low-level network protocol that treats every packet as a live object, and every router as a host where that packet can run. Classes are transferred to a node only once; the packets are just instances of packet classes that can tweak their passage through a network, and even perform real computation on the way.

5.2.2. E

The *E* platform [18] supports fault-tolerant, secure, distributed computing. The system is designed to be secure first, with the other features added only in ways that do not compromise security. As a result, synchronization, threading and communication are not directly accessible. Furthermore, the concepts of mailboxes and behaviors that are so prevalent in the Java-based actor systems at OSL are not exposed to the developer. *E* defines its own language which, while technically derived from Java, is heavily modified to support several new paradigms, including functional programming, capability-based programming, distributed computing, and security. [24] Most of these are beyond the scope of this document, but the mechanisms supporting distributed computing can be compared.

In actor systems, an actor is defined as a tuple consisting of a behavior, a thread, and a mailbox. The mailbox is considered the primary object – actors concern themselves with sending messages to mailboxes under the assumption that the appropriate behavior is attached. Assigning a new object to a mailbox in AJ effectively changes the behavior of that actor, as actors only hold references to mailboxes. In *E*, objects are the only things the user can access; the system manages threads and mailboxes automatically. Remote references are for objects, not mailboxes or threads. "Swapping behaviors" is not a meaningful concept in *E*, as a reference to an object is equivalent to a reference directly to a behavior.

Object references can only be accessed by way of another object. In order to find that "first object" which can provide references to the rest of the object graph, *E* provides "sturdy refs" and URIs, much like SALSA. URIs are textual strings that point to a sturdy ref, and sturdy refs are persistent objects held by the system precisely so that outside nodes can retrieve them as their first object. Sturdy refs and URIs are generated and managed by the system, thus it is not possible to reassign a URI to a different object. Indeed, this restriction is one of the foundations of *E*'s security model. Thus even URIs cannot be abused to simulate behavior swapping. Though generated by the system, sturdy refs and URIs are not available for any object by default. The user must carefully choose which objects to expose to the world this way.

Inter-object communication is performed with a peculiar mechanism that eliminates most of the bugs that distributed programs have. Communication is by way of the "*non-blocking promise,*" which consists of an "eventual send" and an optional "when-catch" block. The "eventual send" sends a message asynchronously. If followed by a when-catch block, the message becomes synchronous, with the body of the "when" block executed in a separate thread when the response is received. The "catch" block handles error conditions. There are no mechanisms for synchronizing threads in *E*, once started.

Actors run within a software environment known as a node. In *E*, objects live within vats. While vats seem equivalent to nodes, only vats get threads and mailboxes. Objects use the vat's resources to perform computation and communication. Under this system, migration can only be performed on whole vats. Objects can be cloned into a new vat, but the user must implement both the cloning method, and a mechanism to update references to the old object.

5.2.3. SCALA

SCALA [38] is similar to E in that it re-casts the Java syntax to support functional programming. It retains most of the features of Java, then adds a few like aspect-oriented computing, first-class functions, and parameterized types. [20][22] These new capabilities are the primary focus of SCALA; the intent is to study component-based

software construction. SCALA has no distributed computing facilities built in, however the runtime code includes a library which provides actor classes, much the same way AJ and The Actor Foundry do for plain Java.

SCALA includes a code library implementing a variety of remote communication paradigms, including thread synchronization, futures, and mailboxes. These are all built upon Java's standard tools for threading and synchronizing. SCALA also provides its own “Actor” class, which is built upon the existing threading and mailbox classes. Writing actors in SCALA is no different than in AJ and the Actor Foundry.

5.2.4. Coqa

The main focus of the Coqa project [16] is the inversion of the standard locking model for Java threads: Java supports threads but leaves all synchronization up to the developer, while Coqa has threads acquire locks on first access and hold them until the thread terminates. This greatly reduces parallelism by increasing contention, but Coqa provides new operators that allow the developer to re-introduce parallelism only where it's appropriate [23]. With such aggressive locking, race conditions become impossible, and proof of code correctness becomes much simpler. Unfortunately Coqa provides no mechanism to avoid deadlock, although it claims that static code analysis can easily solve this problem.

Coqa contains no remote access code; it's only purpose is to study parallel computation on shared data. This makes adding remote communication difficult, as remote threads cannot access shared data.

5.2.5. Jsasb (Java Standalone Application Service Bus)

Jsasb [36] is a very preliminary project for studying the notion of “event-driven programming” as a paradigm which builds on object-oriented programming. The Actor model introduced here recalls the very core of the Actor model, but leaves out all of the advanced concepts studied by other Actor-based projects – like remote communication, fair threading, continuations, resource contention, and distributed computing. [35]

6. Analysis

6.1. Summary

AJ was designed with a large number of small classes to make feature management explicit. However, as the number of features grew, the interactions between all the classes began to get complicated. This is not a problem with the design so much as it is a problem with actor systems in general: building a system of multiple independent agents ***is*** complicated.

One of the biggest problems was building name-based addressing as a layer on top of location-based addressing. This is not unusual; a good distributed name service can be a thesis in itself.

As in all things, there's a tradeoff. A more generalized system imposes a heavier run-time load on the CPU, while a more specific system imposes a heavier load on the programmer. And with computers becoming more involved in the day-to-day work of unpredictable humans, static, pre-compiled programs are becoming less efficient. Although a monolithic executable can perform computations at incredible speed, the cost of installing, configuring, and upgrading the executable can be large.

What's needed is a smarter run-time that can remove as much dynamic support from the currently running code as it can, yet still return that behavior if the code changes. Java is a step in the right direction; the just-in-time compiler optimizes code while preserving the dynamic capabilities of Java.

The next step would be to add remote messaging capabilities to the run-time, allowing it to perform message passing or direct function calls as necessary.

6.2. Future Work

AJ is a framework for supporting actors. It contains some basic capabilities, but there are many features that future research can add:

- Add support for futures, which are variables that receive their value at some point in the future as the result of a parallel thread. The owner of the variable can initiate the evaluation, but the owner is not paused until it actually attempts to retrieve the value. Futures are much easier to create and use than callbacks.

- Add support for continuations, which define actor chaining at a high level. This does not necessarily make coding easier, but it makes splitting a computation across methods easier, and increases the amount of parallelism an actor can support. However, the inclusion of futures may render continuations unnecessary.

- Implement fair threading. The Java specification does not require threads to be fair. Thus, it's possible for one actor to consume the entire CPU, and lock out the other actors. Even worse, behavior can vary between operating systems. It is especially critical for Java platforms to behave consistently when actors can migrate, otherwise actors may change their behavior when they move between systems.

- Migrate code along with data. Currently, AJ only supports mobile data – an object instance can move between computers, but the class that defines it cannot. The class definition must be pre-loaded onto all nodes that an actor may run on. Fixing this is almost trivial; in Java, every class is itself an object, and is defined by a single file on disk. To move an actor, AJ would simply send the class file through the network along with the object. Alternately, AJ nodes could be extended to load code from a central web server as needed.

- Implement distributed garbage collection. Currently, the only way to recover the memory used by an actor is to send it a "quit" message. Distributed garbage collection would allow actor management to be as simple as it is with plain Java

objects. Note that distributed garbage collection is more difficult than ordinary pointer management [32]

- Implement distributed name services. A name service is essentially a hash table from strings to network locations. Name services could be built on top of a decentralized distributed hash table to resist node failure.

- Implement a message as an actor. A Message possesses the ability to invoke itself on an actor; a further abstraction would make the message itself into an actor that migrates to where the target actor is. This is essentially what ANTS has done, see section 5.3.1.

- Integrate with Java's built-in serialization facilities. Java includes support for remote method invocation, using built-in serialization facilities to marshal whole objects and duplicate them on a remote system. However, including an actor as a parameter causes the actor to be duplicated on the remote system in a non-runnable state. A more useful trick would be for an Actor to convert itself into an Address whenever anyone attempts to serialize it, allowing the remote program to communicate with the original actor. (Remember, only an Address object is necessary to communicate with an actor; an ActorServer environment is only needed when hosting actors.)

Appendix A. What is Java?

Java is an interpreted, object-oriented language and run-time system that was originally designed for embedded systems, but redesigned to take advantage the recent explosion of interest in the Internet, specifically the World Wide Web. The language is similar to C++, but many features were left out either to simplify the language, or to make it more robust. Some of the features that were removed include:

- Operator overloading was considered syntactic sugar. It can be replaced by explicit method calls, though they're not as elegant.

- Polymorphism was considered to rare to be worth including in Java, as it's rarely used in C++. Java classes can only inherit from one other class, but they can also inherit any number of *interfaces*, each of which can specify method prototypes, but cannot provide any implementations.

- Pointer arithmetic and the "delete" function were replaced with automatic garbage collection. Although reference variables are implemented as pointers, the actual numeric address cannot be accessed. Furthermore, dynamically allocated memory cannot be explicitly returned to the system (an operation know as "delete" in C++). The absence of these two features makes memory corruption impossible -- every reference variable either points to valid memory or contains a NULL, and attempts to use NULL are trapped and prevented.

The Java run-time system is highly self-conscious in that every class is itself an object (an instance of class "Class") and can be manipulated directly. Any object in the system can be queried for the Class that it's an instance of, and the Class can be queried for its name. Java supports a form of reflection where field names and types can be queried at run time, and values can be indirectly changed through this mechanism.

Crashing the Java run-time system is impossible when writing Java programs – errors will cause an exception, which can be trapped and handled gracefully by the program.

Memory is managed by a garbage collector, making memory corruption impossible. Arrays indices are monitored, and illegal values cause an exception. Objects are checked at run-time to verify that they conform to expected classes. If the system crashes, it can only be due to a low-level error in the system – either the compiler, the interpreter, or the libraries that interface with the underlying operating system.

The remainder of this appendix describes the specific features that AJ depends on.

A.1. Dynamic Loading

There's no such thing as a "program" is Java, there is only a "first class." Each class is a separate file on disk, and is loaded only when needed by some executing code. Starting a Java program consists of starting Java, and having it load a class and call the methods that start the program running.

In a traditional Java environment, there are two way to start programs.

- A application can be started from the command line simply by entering "java FooBar". The Java interpreter starts up, loads the file called "FooBar.class", then calls the method called "main", which can then instantiate and call the rest of the program. The main method must have the following prototype:

```
public static void main( String args[] );
```

  - Note that the method is *static* -- Java does not instantiate FooBar before calling main, so main cannot access any of the class's dynamic member variables. Any object can be started from the command line if it has a static main method.

- The "applet" is a kind of program developed for use within browser's on the World Wide Web. An applet is a program with a GUI, but no dedicated window. It's donated a specific piece of real estate from the HTML document it's embedded in, and it is restricted to that space. To become an applet, a class must inherit from class "java.applet.Applet," and must implement four methods the browser can use to control the applet:

init()	Called to inform an applet that it has been loaded. It can load or intialize any other objects, but it should not begin running.
start()	The applet may now begin running normally, accepting and processing input.
stop()	The applet should suspend operation, but not unload. This is usually called when the browser displays a different page. The applet is held in memory in case the user returns to it, at which point start is called again.
destroy()	The applet is no longer desired; it should close and free all resources. Variables should be nulled so that the garbage collector can reclaim the memory.

A.2. Threading

Java has multithreading capabilities built-in. Although not quite as simple as the standard fork/join construct, they're equally capable. In the usual Java vein, a *Thread* is itself an object which can be manipulated, started, and stopped on command. Creating a new thread requires creating a new Thread object, and handing it an instance of a runnable class. To be runnable, a class must inherit from interface Runnable, and implement the following method:

```
public void run();
```

When the thread is started, it will call the "run" method, and will terminate when "run" returns. No other method can be directly started by a thread, but "run" is free to call any methods on any objects it has access to.

There are two mechanisms for suspending and resuming threads. This allows for different models of collision avoidance.

- Wait / Notify – Every object is essentially a monitor, and maintains a lock as well as a queue of threads that desire access to the lock. The method "notify" will wake one thread and give it a chance to acquire the lock, although if it fails, it will be suspended again. The method "notifyAll" wakes all waiting threads, and gives them all a shot at the lock again. "notify" is called automatically whenever the lock

is released. The lock does not protect the object by default, however, any members or methods defined with the keyword "synchronized" will automatically use the lock to permit only one thread access at a time. Any other thread attempting access will be suspended and placed on the queue. If neither the object nor its members are synchronized, the object can be explicitly used as a lock for other threads to use.

- Suspend / Resume – A thread object can be explicitly told to suspend, and will stay that way until another thread tells it to resume. This can be used to control custom resources, like lists that are written by one thread and read by another.

Semaphores are not part of the standard Java library, but can be easily implemented with the above mechanisms.

A.3. Networking

Java includes standardized libraries for working with the Internet, including the ability to create servers and clients. Becoming a client requires only instantiating a suitable Socket or DatagramSocket class, then handing it data to send out. The calling program is blocked until transmission is completed.

A server needs to create a ServerSocket or DatagramSocket. The program will be blocked until communication arrives from a client. DatagramSocket communicates with unformatted blocks of bytes, while Socket/ServerSocket use Java streams for formatting data.

A.4. Serialization

To enhance the utility of streams and sockets, Java 1.1 added the ability to serialize and deserialize objects. *Serialization* is the process of converting a collection of memory objects into a stream of bytes, untangling the pointers between different objects. The objects may contain references to other objects; Java automatically follows the references and serializes everything it finds. Each object is only stored once, however, and a cycle

of objects that point to each other will be stored in a way that allows the original cycle to be rebuilt when the stream is deserialized.

A.5. Introspection

This feature is commonly called “reflection” in Java circles, though it isn’t “true” reflection. Java contains functions to access the members and methods of an object by name at run-time. Normally this can only be done by the compiler at compile-time, but the Java compiler creates Class objects, which are loaded into memory at run-time. As described above, the Class object contains the names of everything in the class, and allows run-time access.

Appendix B. Implementation/Class Reference

AJ is an actor system written in and for Java. AJ presents an extremely modular approach to developing Actor programs: many of the modules are themselves actors. As a result, the system is organized as groups of classes; within each group the classes coordinate heavily to implement a layer of functionality. Lower layers implement message passing and buffering; on top of them are built name services, migration, and the Actor class itself.

Detailed documentation is provided in the source code and in the HTML documentation generated from the source.

B.1. Basic Messaging

Messages are explicit objects in AJ, and AJ contains a framework for handling them.

Message	Basic unit of communication in AJ. Is an interface so that any mechanism can be used to store and dispatch messages. Possesses the "invoke" method which performs the operation coded in the Message on a target object.
MessageBase	Basic implementation of Message: handle return address and synchronous status.
AnyMessage	Simple implementation of Message: Method name is stored as a string; parameters as an array of Object, and *invoke* dispatches the method using Java reflection.
MessageSink	Passive object that receives and processes messages on demand.
MessageSource	Passive object that produces messages on demand.
MessageProxy	Passive object that buffers messages, allowing active producers and consumers (each having their own thread) to communicate.
Response	Wrapper for carrying a return value or exception back to the calling actor.

Responder	Sends a return value back to the calling actor. The default action in ActorServerBase and ActorBase is to wrap the return value or exception in a Response and send it to the caller's mailbox. In general, however, a Responder can do anything.

B.2. Communication

Feeders provide a generalized mechanism for managing multiple protocols with a single actor system. However, only *Address* and *Mailbox* need be extended.

B.2.1. Foundation

Communicating with an actor requires a matched *Address-Mailbox* pair. The mailbox is associated with the actor, and buffers messages. The address can be cloned and distributed to other actors so that they can send messages.

Adresses don't refer to actors; instead they refer to mailboxes. Actors are then told which mailbox to get their messages from. Each actor gets its own mailbox, but this division makes it easier to support migration and the name service.

Address	Sends messages to an actor. Encapsulates the notion of an actor's location. If an actor migrates, the address usually changes as well. Contains methods to send messages to its actor. Adresses can be cloned, shared, and serialized.
Mailbox	Receives and buffers incoming messages on behalf of an actor.
MailboxBase	Basic implementation of a mailbox. Doesn't support any particular communication layer, but contains a queue for buffering messages. Also has a "call" method to stuff an actor's address into outgoing messages so that the receiver knows who sent the message. Can be serialized and moved to another node when its actor migrates.

SimpleResponder	Not used. Mimics the default behaviour of ActorServerBase. Examine the source code to see how to build Responders.

B.2.2. Feeders

The feeder system was added to allow multiple protocols to be used within the same system. Where mailboxes and addresses are paired in the foundation, here feeders are paired with addresses, and only one mailbox class is needed.

Having multiple protocols is handy for optimizing communication: LocalAddress is a fast way to move messages within one address space, while IPAddress can communicate over the Internet.

Feeder	Wraps the network objects to catch incoming messages, and forwards them to a mailbox.
FeederFactory	Creates feeders for a particular protocol.
FActorServer	ActorServer that gives new actors feeders to handle communication.
FMailbox	Mailbox that is feeder-aware. Supports multiple addresses per actor so that several protocols can be used in the same program.

B.2.3. Local (Shared Memory) Messaging

LocalAddress	Address for an actor within the same memory space. Can stuff messages directly into mailboxes with out serializing.
LocalFeeder	Feeder for shared-memory messaging.
LocalFeederFactory	Contructor for LocalFeeder.

B.2.4. TCP/IP Messaging

IPAddress	Address for an actor that listens to a TCP port for messages. Each actor on a system gets its own port number.
IPFeeder	Listens for messages on a TCP port.

IPFeederFactory	Constructor for IPFeeder.

B.2.5. Naming

The naming system was added to provide location independence so that migration could be added. Each actor receives a string name which is registered with a NameServer so that actors can look it up.

Name	A globally unique, location-independant identifier for an actor. Inherits from Address, and automatically manages itself so that the actual address of the actor is always up to date.
NameServer	Maps string names to Addresses. Is itself an actor so that remote actors can look up names.
NameActorServer	ActorServer that gives new actors names, and maintains one ActorServer.

B.3. Actors

Actor	Fundamental remote object in AJ. All actors must implement this interface.
ActorBase	Basic implementation of *Actor* for users to build on. Accepts messages, and calls methods with matching names and parameters.
ActorServer	Fundamental actor manager. Creates and manages actors and their environment.
ActorServerBase	Basic implementation of ActorServer. Capable of creating and activating actors.

B.4. System

Debug	Contains switches for debugging AJ.
AlreadyStartedException	Thrown by an ActorServer when an attempt is made to activate an actor that's already running.
BadActorException	Thrown by ActorServer whenever it fails to create an actor.
BadMessageException	Thrown by communication infrastructure on any attempt to send the wrong type of object through. Generally only Messages and Responses are allowed.

Appendix C. Getting the Source Code

The source code for AJ is available through the OSL research group at the University of Illinois. Below is the address of the home page of OSL.

http://www-osl.cs.uiuc.edu/

The test programs that were benchmarked are included as "Pinger.java" and "PingerCall.java".

Appendix D. Sample Code

D.1. Sample Actor Class

This is a sample actor included with AJ as “Pinger.java”. It operates under any of the included protocols, times how long the messages take to move, and includes an optional custom message class.

```
import osl.aj.*;
import java.net.InetAddress;
import java.util.Date;


/**
 * Simple message-bouncing test.  Tests communication
 * by sending a message and getting a response.
 * Can also test IP layer.
 *
 * <P>
 * <table border width=95%>
 *   <th align=left colspan=2><center>Revision History</center></th>
 *
 *   <tr><td><B                                    >1 Oct 1998</B></td>
 *       <td><B>Version 1.2   "Location Independance"</B></td></tr>
 *
 *   <tr><td><B                                    >12 Jan 1998</B></td>
 *       <td><B>Version 1.1   "Feeders"</B></td></tr>
 *
 *   <tr>
 *     <td align=left valign=top                   >30 Nov 1997</td>
 *     <td align=left valign=top width=80%>
 *       <ul>
 *         <LI>main() removed; use an ActorServer to start this:
 *             <PRE>  java osl.aj.IPActorServer Pinger -s</PRE>
 *         <LI>Options added; for help, run with no parameters:
 *             <PRE>  java osl.aj.IPActorServer Pinger</PRE>
 *       </ul></td>
 *   </tr>
 *
 *   <tr><td><B>24 Oct 1997</B></td>
```

```
 *       <td><B>Version 1.0   "Basic Messaging"</B></td></tr>
 *   <tr>
 *     <td align=left valign=top                >24 Oct 1997</td>
 *     <td align=left valign=top width=80%>
 *       <ul>
 *         <LI>First release.
 *       </ul></td>
 *   </tr>
 *   <tr>
 * </table>
 *
 * @author Bill Zwicky
 * @version 1.2:  Location independance
 */
public class Pinger
  extends ActorBase
{
  // How many times to ping remote actor (numbered 1..QTY)
  static final int QTY = 500;

  // Which ping to start the timer on.
  // Should be >1 to allow the classes to load and start.
  static final int FIRST_TIME = 5;

  // Use custom message class?  else use AnyMessage
  static final boolean CUSTOM_MESSAGE = false;

  // Print a line for each ping
  static final boolean PRINT_PROGRESS = true;


  String _name;


  /**
   * Default constructor.  Does nothing.
   */
  public Pinger()
    {
    }


  /**
   * Message that corresponds to "main".
   * FActorServer calls this when started from the command line.
```

```
   *
   * @param args Command line parameters for this actor.
   */
  public void startApp( String args[] )
    {
      int i;
      Object val;
      Address target;
      Address serv = getCurrentMessage().getSource();
      Address rserv = null;
      long start=0, end;
      double rate;


      // I'm "a", other end will be "b"
      init( "a" );

      // Two-server mode; synthesize an address and start b over there
      if( args.length == 2 )
        {
          try
            {
              rserv = new IPAddress(
                InetAddress.getByName( args[0] ),
                Integer.parseInt( args[1] )
                );
            }
          catch( java.net.UnknownHostException ex )
            { }
        }

      // I don't understand; print instruction and exit
      else if( args.length != 1 || !args[0].equalsIgnoreCase("-s") )
        {
          System.out.println( "Usage:" );
          System.out.println( "  <server> Pinger -s" );
            System.out.println( "     ->To run stand-alone demo using any comm
layer" );
          System.out.println( "  ip Pinger <ip address> <port>" );
           System.out.println( "     ->To use remote IPActorServer at <port> on
machine <ip address>" );
          System.exit( 1 );
        }

      try
```

```
{
  // Create the other actor on an appropriate server.
  // I could fake it by saying 'rserv=serv' above,
  // but I prefer to keep them separate so I know
  // who to shut down below.
  if( rserv == null )
    target = (Address)call( serv, "create", "Pinger" );
  else
    target = (Address)call( rserv, "create", "Pinger" );

  // Other end is "b"
  send( target, "init", "b" );

  // Ping b a few times
  for( i=1; i<=QTY; i++ )
    {
      // Start the clock
      if( i == FIRST_TIME )
        start = System.currentTimeMillis();

      if( PRINT_PROGRESS )
        System.out.print( "Pinging " + i + ";" );

      if( CUSTOM_MESSAGE )
        val = call( target, new PingMessage(i) );
      else
        val = call( target, "ping", new Integer(i) );

      if( PRINT_PROGRESS )
        System.out.println( " got " + val );
    }

  // Stop the clock
  end = System.currentTimeMillis();


  //DON'T kill remote server; user might use it again
  send( target, "quit" );         //kill remote actor
  send( serv, "quit" );           //kill our server
  send( getAddress(), "quit" );   //kill us

  // Computing timing
  rate = (end - start)
         / 1000.0
         / (QTY - FIRST_TIME + 1);
```

```
          System.out.println(
              "** Message time:  " + rate + " s/message\n"
            + "** Message rate:  " + 1/rate + " message/s\n"
            + "Done!" );
        }
      catch( Exception e )
        {
          e.printStackTrace();
        }
    }

  /**
   * Message that corresponds to constructor.
   */
  public void init( String n )
    {
      _name = n;
      System.out.println( _name + " is active as " + getAddress() );
    }

  /**
   * The test method.
   */
  public Integer ping( Integer n )
    {
      return new Integer( n.intValue() + 100 );
    }

  /**
   * Print info before quitting.
   */
  public void quit()
    {
      System.out.println( _name + " quitting" );
      super.quit();
    }
}


class PingMessage
  extends MessageBase
{
  //The number to pass to ping()
  int _val;
```

```
  //Needed for getTemplate(), but not actually used.
  static transient Class _templ[];
  static
    {
      _templ = new Class[1];
      _templ[0] = Integer.class;
    }

  public PingMessage( int i )
    {
      _val = i;
    }

  //these are not actually needed, but the interface calls for them
  //(Should it?)
  public String   getMethod()
    { return "ping"; }
  public Class[]  getTemplate()
    { return _templ; }

  /**
   * Call method "ping" on a target object.  Object must be of
   * class "ping", else ClassCastException will be thrown.  Doing
   * this saves the overhead of Java reflection incurred by the
   * generic AnyMessage.
   */
  public Object invoke( Object target )
    throws Exception
    {
      return ((ping)target).ping(new Integer(_val));
    }
}
```

D.2. Migration Test Class

```
import osl.aj.*;
import java.net.InetAddress;


/**
 * Same test as "Pinger.java", but creates remote actor using moveTo().
 * To run alone, use:
 *   <PRE>  name namer -s</PRE>
 * To run with other servers, start those servers with just:
 *   <PRE>  name</PRE>
 * then enter:
 *   <PRE>  name namer {&lt;host&gt; &lt;port&gt;}+</PRE>
 * and give the hostname and port number for each server you started.
 *
 * <P>
 * <table border width=95%>
 *   <th align=left colspan=2><center>Revision History</center></th>
 *
 *   <tr><td><B                                    >1 Oct 1998</B></td>
 *       <td><B>Version 1.2   "Location Independance"</B></td></tr>
 *
 *   <tr><td><B                                    >12 Jan 1998</B></td>
 *       <td><B>Version 1.1   "Feeders"</B></td></tr>
 *
 *   <tr>
 *     <td align=left valign=top                   >4 Jan 1998</td>
 *     <td align=left valign=top width=80%>
 *       <ul>
 *         <LI>Created.
 *       </ul></td>
 *   </tr>
 * </table>
 *
 * @author Bill Zwicky
 * @version 1.2:  Location independance
 */
public class mover
  extends ActorBase
{
    //for both:
  String  _name;         //string id for user's benefit; has nothing to do with
Names
    //for static app:
```

```
Address _target;       //address for roving actor
  //for roving app:
Address _servers[];    //list of servers to bounce between
int     _currServer;   //which _server[] we're at
Address _defer;        //DeferredDeliverer for timed messaging


public mover()
  {
  }

/**
 * Start up a roving actor (another <B>mover</B>) and throw 20 pings
 * at it.  Pinging is done by a separate recursive doPing method
 * so that this actor can respond to getServer() from the Name updators.
 * (ForwardUpdator is a secret actor created by ForwardActor to fix up
 * NameServers.)
 */
public void startApp( String args[] )
  {
    int i;
    Address serv = getServer();
    Address rserv = null;

    init( "a" );

    if( args.length > 0  &&  (args.length % 2 == 0) )
      {
        try
          {
            _servers = new Address[args.length / 2 + 1];
            _servers[0] = serv;

            for( i = 0;  i < args.length/2;  i++ )
              {
                _servers[i+1] = new IPAddress(
                    InetAddress.getByName( args[i*2] ),
                    Integer.parseInt( args[i*2+1] ) );
              }
            _currServer = 0;
            rserv = _servers[1];
          }
        catch( java.net.UnknownHostException ex )
          {
            ex.printStackTrace();
```

```
            System.exit( 5 );
          }
        }

      else if( args.length != 1 || !args[0].equalsIgnoreCase("-s") )
        {
          System.out.println( "Usage:" );
          System.out.println( "  <server> mover -s" );
            System.out.println( "     ->To run stand-alone demo using any comm
layer" );
          System.out.println( "  ip mover {<ip address> <port>}+" );
           System.out.println( "     ->To use remote IPActorServer at <port> on
machine <ip address>" );
          System.exit( 1 );
        }

      try
        {
          // Create the other actor here; it will move when we ping it
          _target = (Address)call( serv, "create", "mover" );
          send( _target, "init", "b" );

          if( _servers != null )
            {
              //send list of servers, and an int telling where it currently is
              send( _target, "init2", _servers, new Integer(0) );
              //send( target, "moveTo", _servers[1] );
            }

          _defer = (Address)call( serv, "create", "osl.aj.DeferredDeliverer" );

          //get pinging started
          send( getAddress(), "doPing", new Integer(1) );
        }
      catch( Exception e )
        {
          e.printStackTrace();
        }
    }



  public void init( String n )
    {
      _name = n;
```

```
    System.out.println( _name + " is active as " + getAddress() );
  }



/**
 * Init the roving actor.
 * @param servs Array of servers to move between.
 * @param cur   Index in servs[] that this actor is currently on.
 */
public void init2( Address[] servs, Integer cur )
  {
    _servers = servs;
    _currServer = cur.intValue();
  }



/**
 * Send the ping messages; shut down when enough have been sent.
 * Implemented "recursively" so that this actor can respond to
 * getServer() from the Name updators.
 */
public void doPing( Integer n )
  {
    int i = n.intValue();
    Object val;

    try
      {
        if( i <= 20 )
          {
            //do this ping
            System.out.print( "Pinging " + i + ";" );
            System.out.flush();
            val = call( _target, "ping", new Integer(i) );
            System.out.println( " got " + val );

            //DeferredDeliverer allows this actor to process getServer msgs
            System.out.println( "  Waiting for actor to move.." );
            send( _defer, "defer",
                new Float(4),                              //after 4 secs ..
                getAddress(),                              //tell myself ..
                new AnyMessage("doPing", new Integer(i+1))  //to keep pinging
                );
```

```
            }

          else
            {
              Address serv = getServer();

              //DON'T kill remote server; user might use it again
              send( _defer, "quit" );          //kill DeferredDeliverer
              send( _target, "quit" );         //kill remote actor
              send( serv, "quit" );            //kill our server
              send( getAddress(), "quit" );    //kill us

              System.out.println( "Done!" );
            }
        }
      catch( Exception e )
        {
          e.printStackTrace();
        }
    }



  /**
   * Respond to a message.  Returns the parameter + 100, then moves
   * this actor to the next server in the list.
   */
  public Integer ping( Integer n )
    {
      //queue a request to move
      try
        {
          _currServer = (_currServer + 1) % _servers.length;
          if(n.intValue()<=2)
          send( getAddress(), "moveTo", _servers[_currServer] );
        }
      catch( Exception ex )
        { }
      return new Integer(n.intValue()+100);
    }
}
```

Appendix E. References

[1] Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Agha, G. and Hewitt, C. *Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism.* Technical Report. UMI Order Number: AIM-865., Massachusetts Institute of Technology, 1985.

[3] Agha G. and Kim W. *Actors: A Unifying Model for Parallel and Distributed Computing.* In Journal of Systems Architecture, vol. 45, no. 15, Sep. 1999, pp. 1263-1277.

[4] Armstrong, J.; Virding, R.; Wikström, C.; and Williams, M. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[5] Astley, Mark. *The Actor Foundry: A Java-based Actor Programming Environment*, published 2000. Retrieved 2007 from <http://osl.cs.uiuc.edu/foundry/>.

[6] Ayres, John W.; and Eisenbach, Susan. *Stage: Python with Actors*. Published 2008, retrieved 2008 from <http://pubs.doc.ic.ac.uk/actors-in-python/>.

[7] Briot, Jean-Pierre. *From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80*. In Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming, San Diego, California, 1988, pp. 69-72.

[8] Briot, Jean-Pierre. *About Actalk*. Published 2007, retreived 2008 from <http://www-poleia.lip6.fr/~briot/actalk/actalk.html>.

[9] Chang, Po-Hao. *Customizing Web Applications Through Adaptable Components and Reconfigurable Distribution*. PhD Thesis, University of Illinois at Urbana-Champaign, 2007.

[10] Chien, Andrew A. *Concurrent Aggregates (CA): Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, Massachusetts, 1993.

[11] Houck, Chris and Agha, Gul. *HAL: A High-level Actor Language and Its Distributed Implementation*. In 21st International Conference on Parallel Processing (ICPP '92), St. Charles, IL, August 1992, volume II, pp. 158-165.

[12] Jang, Myeong-Wuk. *The Actor Architecture*, published 2004. Retrieved 2008 from <http://osl.cs.uiuc.edu/aa/>.

[13] Kafura, Dennis; and Lee, Keung Hae. *ACT++: Building a Concurrent C++ with Actors*. Journal of Object-Oriented Programming, 1990, Vol. 3, No. 1, pp. 25-37.

[14] Kim, WooYoung. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD Thesis, University of Illinois at Urbana-Champaign, 1997.

[15] Kim, WooYoung and Agha, Gul. *Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages*. In Proceedings of Supercomputing '95, IEEE, 1995.

[16] Lu, Xiaoqi. *Coqa: A Concurrent Programming Model with Ubiquitous Atomicity*. PhD thesis, Johns Hopkins University, 2007.

[17] Masuhara, Hidehiko. *The ABCL family of languages*. Published 1994, retrieved 2008 from <http://web.yl.is.s.u-tokyo.ac.jp/pl/abcl.html>.

[18] Miller, Mark S., et al. *Welcome to ERights.Org*, published 2008. Retrieved 2008 from <http://www.erights.org/>.

[19] Nielsen, B. and Agha, G. *Towards Reusable Real-Time Objects.* In Annals of Software Engineering: Special Volume on Real-Time Software Engineering, vol 7, no. 1-4, pp. 257-282, Oct 1999, Springer Netherlands.

[20] Odersky, Martin; Altherr, Philippe; et. al. *An Overview of the Scala Programming Language*, published 2006. Retrieved 2008 from <http://www.scala-lang.org/docu/files/ScalaOverview.pdf>.

[21] Ren, S.; Venkatasubramanian, N.; and Agha, G. *Formalizing Multimedia QoS Constraints Using Actors.* In Proceedings of the Second IFIP International Conference on Formal Methods for Open, Object-Based Distributed Systems, Canterbury, United Kingdom: Chapman & Hall, Ltd., 1997, pp. 139-153.

[22] Schinz, Michel and Haller, Philipp. *A Scala Tutorial for Java Programmers*, published 2008. Retrieved 2008 from <http://www.scala-lang.org/docu/files/ScalaTutorial.pdf>.

[23] Smith, Scott. *Coqa: Concurrent Objects with Quantized Atomicity*, published 2007. Retrieved 2008 from <http://www.cs.jhu.edu/~scott/pll/coqa.html>.

[24] Stiegler, Marc. *The E Language in a Walnut*, published 2007. Retrieved 2008 from <http://wiki.erights.org/wiki/Walnut/Complete>.

[25] Sturman, Daniel. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[26] Sturman, D. and Agha, G. *A Protocol Description Language for Customizing Failure Semantics.* In The 13th Symposium on Reliable Distributed Systems (Dana Point, CA), 1994, pp. 148-157.

[27] Taura, Kenjiro; Matsuoka, Satoshi; and Yonezawa, Akinori. *ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language – Its Design and Implementation*. In Proceedings of the DIMACS workshop on Specification of Parallel Algorithms, 1994.

[28] Tomlinson, C., Kim, W., Scheevel, M., Singh, V., Will, B., and Agha, G. 1988. *Rosette: An Object-Oriented Concurrent Systems Architecture.* In Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming (San Diego, CA). G. Agha, P. Wegner, and A. Yonezawa, Eds. ACM, New York, NY, pages 91-93.

[29] Varela, Carlos. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[30] Varela, Carlos. *The SALSA Programming Language*, published 2007. Retrieved 2008 from <http://www.cs.rpi.edu/research/groups/wwc/salsa/>.

[31] Varela, C. and Agha, G. *What After Java? From Objects to Actors.* In Computer Networks and ISDN Systems, Volume 30, Number 1, April 1998, pages 573-577.

[32] Venkatasubramanian, N.; Agha, G.; and Talcott, C. *Scalable Distributed Garbage Collection for Systems of Active Objects.* In International Workshop on Memory Management (St. Malo, France), 1992, pages 134-147.

[33] Wetherall, David, Guttag, John, and Tennenhouse, David. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols*. IEEE OPENARCH'98, San Francisco, CA, April 1998.

[34] Wetherall, David and Whitaker, Andrew. *Active Networks at UW*, published 2005. Retrieved 2005 from <http://www.cs.washington.edu/research/networking/ants/>.

[35] Young, Rex. *Jsasb Adds an Actor Model Event-Driven Programming Paradigm to Java*, published 2007. Retrieved 2008 from <http://java.sys-con.com/read/464425.htm>.

[36] Young, Rex. *Jsasb Project Home*, published 2007. Retrieved 2008 from <https://jsasb.dev.java.net/>.

[37] *Open-Source Erlang*, published 2008. Retrieved 2008 from <http://www.erlang.org/>.

[38] *The Scala Programming Language*, published 2008 by the Programming Methods Laboratory at École Polytechnique Fédérale de Lausanne. Retrieved 2008 from <http://www.scala-lang.org/>.