AiR: Actor inspired Ruby
MEng Individual Project Report

Alisdair Johnstone

Supervisor: Susan Eisenbach
Second marker: Emil Lupu

Imperial College London

June 27, 2008
Abstract

The last few years have seen a fundamental shift towards multi-core systems. This is true for desktops, where two or four cores are standard, and especially true for servers, where as many as thirty two or sixty four cores may be resident on a single processor.

It is important the new found potential for parallelism afforded by this new generation of processors is fully exploited; this requires that software developed for such systems be written with concurrency in mind. Traditional concurrency abstractions such as threads have proven to be cumbersome to work with, and the requirement that explicit locking constructs be used is often a source of error.

Carl Hewitt’s actor model has been suggested as a better abstraction to dealing with the issue of concurrency. This report details the development of AiR, a modern actor-based domain specific language and supporting runtime environment that can used by Ruby programmers. AiR aims to provide a rich feature set, such as built in support for distribution and mobility, and at the same time tries to retain the elegance and ease-of-use that Ruby has come to be known for.
Acknowledgements

I would first like to give special thanks to my supervisor, Professor Susan Eisenbach, for her continual support and encouragement throughout the course of the project. Secondly, I would also like to thank my second marker, Dr Emil Lupu, for providing feedback on an early version of the project. Additionally I would like to thank John Ayres for answering the multitude of questions I sent him, and for suggesting a number of sample applications.

Lastly, I would like to thank my parents who have always given their full support throughout my time spent at university.
# Contents

1 Introduction .................................................. 1
   1.1 Contributions ............................................ 2

2 Background .................................................. 3
   2.1 Actor Model .............................................. 3
   2.2 Ruby Programming Language ............................. 5
      2.2.1 Language Introduction .............................. 5
      2.2.2 Concurrency Features ............................... 6
   2.3 Actor Languages & Libraries ............................. 9
      2.3.1 Erlang ............................................... 9
      2.3.2 Scala ............................................... 12
      2.3.3 SALSA ............................................. 15
      2.3.4 Stage ............................................. 18
      2.3.5 Ruby: Erlectricity ................................. 19
      2.3.6 Ruby: Omnibus Concurrency Library ................ 20

3 Specification ............................................... 23
   3.1 Integration with Ruby .................................... 23
   3.2 Message Passing Abstraction ............................ 23
   3.3 Distribution ............................................ 23
   3.4 Mobility ................................................ 24
   3.5 Scalability .............................................. 24
   3.6 Security ................................................ 24
4 Implementation

4.1 Runtime Overview ............................................. 25
4.2 Theatre ............................................................. 26
  4.2.1 Master Theatre ............................................. 27
  4.2.2 Slave Theatre ............................................. 29
  4.2.3 Theatre Design Choices ................................. 31
4.3 Actors ............................................................... 32
  4.3.1 Achieving Concurrent Execution ......................... 32
  4.3.2 Message Passing Abstraction ............................ 33
4.4 Mobility ............................................................. 37
4.5Naming ............................................................... 38
4.6 Summary ............................................................. 39

5 Language

5.1 Getting started ................................................... 41
5.2 Actors ............................................................... 41
  5.2.1 Behaviour definition ...................................... 41
  5.2.2 Instantiation .................................................. 42
5.3 Theatres ............................................................. 43
5.4 Message passing .................................................. 43
  5.4.1 Two-way messaging ....................................... 43
  5.4.2 One-way messaging ....................................... 44
  5.4.3 Priority message passing ................................. 44
  5.4.4 Composite message configuration ...................... 45
5.5 Futures ............................................................... 45
  5.5.1 Synchronization ............................................. 45
  5.5.2 Status Check ................................................. 46
  5.5.3 Selective Wait ................................................ 46
5.6 Message Chains ................................................... 47
  5.6.1 Specification of a chain .................................. 47
  5.6.2 Further uses of message chains ....................... 49
5.7 Dynamic Interfaces ............................................... 50
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7.1 Receive Blocks</td>
<td>50</td>
</tr>
<tr>
<td>5.7.2 Chords</td>
<td>53</td>
</tr>
<tr>
<td>5.8 Naming</td>
<td>55</td>
</tr>
<tr>
<td>5.8.1 Specifying a nameserver</td>
<td>55</td>
</tr>
<tr>
<td>5.8.2 Standard naming operations</td>
<td>55</td>
</tr>
<tr>
<td>5.8.3 Capability Tags</td>
<td>56</td>
</tr>
<tr>
<td>5.9 Migration</td>
<td>57</td>
</tr>
<tr>
<td>5.10 Exception Handling</td>
<td>58</td>
</tr>
<tr>
<td>5.10.1 Tradition approach</td>
<td>58</td>
</tr>
<tr>
<td>5.10.2 Linked Actors</td>
<td>58</td>
</tr>
<tr>
<td>5.11 Summary</td>
<td>59</td>
</tr>
<tr>
<td>6 Evaluation</td>
<td>62</td>
</tr>
<tr>
<td>6.1 Correctness</td>
<td>62</td>
</tr>
<tr>
<td>6.2 Performance</td>
<td>64</td>
</tr>
<tr>
<td>6.2.1 Execution Speed</td>
<td>64</td>
</tr>
<tr>
<td>6.2.2 Scalability</td>
<td>67</td>
</tr>
<tr>
<td>6.3 Effectiveness</td>
<td>69</td>
</tr>
<tr>
<td>6.3.1 Distributed Chat Application</td>
<td>70</td>
</tr>
<tr>
<td>6.3.2 News website</td>
<td>71</td>
</tr>
<tr>
<td>6.4 Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>75</td>
</tr>
<tr>
<td>7.1 Further Work</td>
<td>77</td>
</tr>
<tr>
<td>7.2 Closing remarks</td>
<td>78</td>
</tr>
<tr>
<td>A Syntax</td>
<td>83</td>
</tr>
<tr>
<td>A.1 Scala</td>
<td>83</td>
</tr>
<tr>
<td>A.2 SALSA</td>
<td>84</td>
</tr>
<tr>
<td>A.3 AiR API &amp; Grammar</td>
<td>84</td>
</tr>
<tr>
<td>A.4 Monte Carlo π approximation</td>
<td>87</td>
</tr>
<tr>
<td>A.4.1 Sequential Ruby Version</td>
<td>87</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The traditional computing landscape has recently undergone a dramatic transformation. Whilst just a few years ago it was taken as a given that a typical home computer would have a single processor core, this is no longer the case - with dual-core and quad-core processors now being the norm. In this years to come it is highly likely that an exponential increase in cores will be seen, just as how we had been used to seeing exponential increases in clock rates for single core systems before the shift.

Although the ability to execute applications with true parallelism is certainly welcome, it has left a fundamental problem for the software industry which for too long has simply ignored the issue of concurrency. When the issue is dealt with, the standard approach is to build applications that consist of multiple threads that communicate with each other through items in shared memory. To ensure that integrity of such shared data items is maintained, programmers must resort to using low level locking mechanisms such as semaphores and monitors. When such mechanisms are not handled correctly it is extremely easy to run into problems such as deadlocks and race conditions.

The design of computationally intensive software need not be bound by the local processing power available on any one machine, but indeed with innovations such as the Internet being in such wide spread usage it is possible to envisage software that is distributed between thousands or even millions of systems.

The actor model, first proposed by Carl Hewitt, Peter Bishop, and Richard Steiger [41], provides a model of computation that inherently exploits parallelism and that is well suited for distributed environments. Actors are autonomous entities that operate concurrently and communicate using asynchronous message passing. Actors each have their own unique behaviour, and posses a private data store.

Advantages of using the actor model over a shared memory threaded approach are copious, and include: increased scalability, built-in concurrency, and the
removal of the need for low level locking mechanisms such as semaphores and monitors. Section 2.1 gives a more in depth discussion of the model and its benefits.

To date there have been numerous implementations of the actor model, with the most well know example being Erlang. It is unfortunate however that these languages are typically not friendly towards object-oriented programmers, and as such are oft overlooked by the majority. Another pitfall is that scalability is often not fully considered, with languages often resorting to assigning actors heavyweight native threads or even entire system processes. An in-depth overview of the state-of-the-art in actor languages is given in section 2.3.

1.1 Contributions

The core contribution of this project is the development of a new object-oriented actor-based library for Ruby; this library, known as AiR, provides a domain specific language for writing concurrent applications and a supporting runtime environment for such applications. The aim of AiR is to make available the actor model in a manner accessible to object-oriented programmers, avoid the pitfalls of other similar efforts, and at the same time offer a rich feature set that can be used in a clear and intuitive manner.

Ruby was chosen because of its pure object-oriented semantics and the inherent beauty and conciseness of the code that is produced by those well versed in the language. The author of this report has had several encounters with the language before and can attest these claims. Section 2.2 gives a fuller discussion of the Ruby programming language and the standard concurrency primitives that it offers.

AiR provides in-built support for distribution and mobility, and provides a number of novel approaches to tackling problems such as interface restriction. The design and implementation details of AiR are discussed in chapter 4, and an overview of the language from a developers perspective is given in chapter 5.

Besides creating AiR itself, numerous applications were built using the language. These examples allowed AiR to be tested as development progressed, highlighting failing and areas of improvement. They should also prove indispensable to any programmer wishing to try AiR, and as such are included with the library. In chapter 6 a number of these sample applications are discussed for the purposes of evaluating AiR’s performance and effectiveness as a language.

Chapter 7 presents the conclusions of the project and details a number of areas that could be looked at to expand and improve upon the current implementation.
Chapter 2

Background

2.1 Actor Model

The Actor model is a concurrent model of computation first introduced by Carl Hewitt [40, 41] in the 1970s and later developed Gul Agha [17] and others. In recent years there has been renewed interest in the model due to the current trend towards multi-core and multi-processor systems.

The basic unit of computation in the Actor model is referred to as an ‘actor’. Actors are self-contained entities that communicate through asynchronous (non-blocking) message passing. Each actor has its own unique thread of control, providing for concurrent execution of actors, and an associated behaviour which defines how it is to act in response to incoming messages. Actors do not share any state between themselves, and hence a number of problems commonly associated with shared memory systems can be avoided.

The behaviour of an actor, as outlined in [19], can be defined through a combination of the following actions (forming a ‘script’ [44]):

- Sending messages to acquaintance actors (send)
- Creating new actors (create).
- Specifying its replacement behaviour (become).

The first two actions are self explanatory. The become primitive allows an actor to define a new actor which will process the next message that arrives (with the original actor forwarding communications to the new actor); this allows message processing to be pipelined, thus increasing potential parallelism.

Actors each have a unique ‘mailing address’ which is used to identify an actor when performing a send. There are three methods by which an actor can become acquainted with another actor (i.e. learn its mailing address), namely:
• Receiving the address in an incoming message.
• Being passed the address as part of initialization.
• Creating the other actor using the create action.

Actor systems constantly undergo a process of evolution; at any point in time actors may be added to the system, removed from the system, or change their acquaintances. The term ‘open systems’ [42] is used in the actor community to refer to such dynamic topologies.

In the world of actor theory, everything is an actor. Three types of actors exist, as outlined in [31]:

**Primitive Actors** Such actors are used to model data item (a character for example) and procedure primitives, providing a way to bottle-out computations.

**Unserialized Actors** These are actors which maintain a fixed local state. An example of such an actor is an actor which calculates factorials.

**Serialized Actors** Unlike their brethren, serialized actors can modify their local state. An example of such an actor is an actor which represents a bank account.

Due to the fact that the actor model was first created in the 1970s - the age of the mainframe, the initial version did not focus on aspects of distribution or mobility. The model does however scale naturally for both of these, and as a result many recent implementations have support for one or both.

The following list highlights some of the advantages of the actor model over other approaches to concurrency:

• Actors are inherently concurrent.
• An actor can be conceptualized as the fusion of an object and a thread, and therefore control flow of the application need not be considered separately from actors themselves.
• The model is highly scalable, as exemplified by Erlang (2.3.1).
• Low-level locking constructs such as semaphores and monitors are not needed, with locking being built into the communication model.
• Asynchronous messaging increases parallelism, due to the fact that it removes a source of blocking.
• There are no low level race conditions, since each message is processed atomically.
• It is more representative of the real world than other models [20].
2.2 Ruby Programming Language

This section will give a brief introduction to the Ruby language, followed by a
discussion of the standard approach to concurrency used by Ruby programmers
at present.

2.2.1 Language Introduction

Ruby was created by Japanese programmer Yukihiro “Matz” Matsumoto in
1993. Outside of its native Japan the language was relatively unknown until
around the turn of the century when it began rising to prominence elsewhere.
In the last few years especially the language has seen an explosion of growth –
due in part to the Ruby on Rails web framework.

The language itself follows a dynamic, pure-object oriented approach; every-
thing is an object, including: classes, methods, integers, floats, etc. Functional
features such as closures, anonymous functions, and continuations are also made
available – as with many other dynamic languages; this mix works very well, as
will later be demonstrated by some examples.

Ruby offers extremely powerful meta-programming support as standard. Op-
erations such as adding new methods or classes at runtime can be achieved
with ease. Likewise, introspection – listing all methods offered by an object for
example – is extremely simple. This facet in particular has meant that Ruby
has been a popular choice for the implementation of embedded domain specific
languages, allowing developers to easily extend and modify the original seman-
tics of Ruby to suit a particular domain. Active Record, the object-relationship
mapping package included with Rails, is a particularly prominent example in
which such techniques are employed for the purpose of database interaction [50].

Blocks (closures) are a widely used and much cherished feature of Ruby. The
primary purpose blocks serve is to act as a callback mechanism from a called
method to the calling context; the called method need not terminate to do this
however, it simply has to call the 'yield' method on the block, and arguments
can additionally be given to the block as part of the yield call. The following
simple example demonstrates a trivial usage of blocks:

```ruby
1 def each_country
2   print "-› Inside each_country"
3   yield("Japan")
4   yield("England")
5   print "-› Exiting each_country"
6 end
7
8 greeting = "Hello"
9 each_country { |country| print "#{greeting} #{country}"}
```

Executing the above code will result in the following output:
-> Inside each_country
Hello Japan
Hello England
-> Exiting each_country

Description: After the each_country method has been defined, a call to the method is made, passing in the code block `{|country| print “Hello #{country}”}` as an (implicit) argument. Once inside the method a regular call to print is made, printing the text “-> Inside each_country”.

The next line is more interested, here the yield method is called, passing in the text “Japan”, and at this point control is passed to the block, with the given text being bound to the block argument “country”. Inside the block we print out the concatenation of the greeting object (note how this originates from the caller’s context) and the country object.

Control now returns to the “each_country” method, where the process is repeated with a different string, before finally printing “-> Exiting each_country” and returning. Blocks provide an elegant approach to common tasks such as iteration, and also simplify tasks such as reading files - the read method opens the file descriptor, returns it as a block argument, and then closes it once the block returns (thus freeing the programmer from explicit opening or closing).

This final example - mimicking the first - demonstrates the use of blocks with the built-in ‘each’ method defined upon Arrays:

```ruby
["Japan", "England"].each {|country| puts "Hello #{country}"}
```

Ruby’s light-weight programming style, combined with its rich and powerful feature set make it an excellent language choice for rapid (and enjoyable) development. The main drawback of Ruby is that it is a directly interpreted language, and as such can have rather slow execution speeds, as compared to Java say - not a problem if being used prototyping purposes; this is however changing, with an official migration to an alternative virtual machine “YARV” planned. YARV first compiles code to a byte-code format as to speed up execution.

### 2.2.2 Concurrency Features

Like most other modern programming languages, the two primary abstractions offered by Ruby for concurrency are threads and processes.

**Threads**

Threads are a mechanism in which several streams of control (an instruction pointer plus a stack) can be specified for a single program instance. When
a thread is executing a piece of code that accesses a shared area of memory, it is important to ensure mutual exclusion through use of constructs such as semaphores or monitors.

Ruby specifically offers what are known as a ‘green threads’. Green threads, unlike native threads, are scheduled by the Ruby virtual machine itself, as opposed to being scheduled by the operating system. The advantage of this approach is that context switching is a relatively light-weight operation, and relatively little memory is used up per thread. The disadvantage of green threads is that each Ruby virtual machine instance resides on a single CPU core, and as such any chances for true parallelism are lost.

The below code shows the creation of a thread which prints out the numbers one through ten before terminating.

```ruby
Thread.new do
  1.upto(10) {'puts i'}
end
```

The standard construction method ‘new’ is called on the Thread class, passing in a code block – using the do/end alternative to curly bracket notation - which represents the code to execute when the thread is started. Any method or variable available to the context in which the thread was created will also be available within the code block passed to the thread – as demonstrated in previous code block examples.

Classes are additionally available for the creation of semaphores, monitors, and the like. These constructs may also be ‘mixed-in’ to other classes as a means to ensure mutual exclusion of method execution within instances of such classes.

**Processes**

Processes are similar to threads in that they each represent a unique stream of control. They differ in that processes share nothing with each other – whereas threads share code and data. A process is in fact nothing more than a program instance with a single thread.

Since processes do not share memory with each other there is no need to try and ensure mutual exclusion to protect against shared memory violations.

Processes in Ruby can be created through calling built-in methods of the Process module, such as ‘fork’ - which creates a duplicate of the current process. See [http://ruby-doc.org/core/classes/Process.html](http://ruby-doc.org/core/classes/Process.html) for more details about this module.

Each Ruby process has its own virtual machine instance. This has the advantage that true parallelism can be achieved by executing multiple processes on several cores at once. The disadvantage of this is that processes are extremely heavy weight – instances of the Ruby virtual machine use a lot of memory.
Conclusion

Concurrency in Ruby is a tricky issue, with neither processes nor threads offering an ideal solution. The advantages and disadvantages of each are the mutual inverses of the other, and therefore the actor library which is to be built on top of these constructs should aim for an intersection of the positive aspects of each, while at the same time attempting to minimize or eliminate the negative aspects.
2.3 Actor Languages & Libraries

The discussion shall now turn focus to the current generation of actor languages and libraries. Some have been tried and tested in industry settings, while others are more experimental in nature. To be genuinely useful and usable, all of the languages and libraries work at a higher level of abstraction than the pure actor model introduced in section 2.1. Many interesting and innovative features are offered, and these will act as inspiration for AiR’s own feature set.

2.3.1 Erlang

Erlang is a concurrent functional programming language designed for the creation of real-time, distributed fault-tolerant systems [23].

The language was initially developed at Ericsson to investigate the suitability of declarative programming in the context of developing telecommunication switching devices. The original prototype interpreter for Erlang was written in Prolog; this however was later changed for efficiency purposes and because the language was moving more in the direction of the functional programming style [28].

In 1998 Erlang was released as an open-source project, making it available to the wider world outside its native environment at Ericsson. Since this time it has enjoyed moderate success, being used at companies such as Amazon, Nortel, and T-Mobile [4]. There is currently much talk about Erlang in the development community as a whole, and just recently the Pragmatic Bookshelf published a new book “Programming Erlang, Software for a Concurrent World” [22] by Joe Armstrong, one of the original creators of Erlang.

Light-weight Processes

Erlang has no inherent notion of an actor, instead its currency of concurrency comes in the form of a compatible concept known as processes.

Processes represent independent threads of execution. The difference between processes and regular threads however is that the former share no state and instead communicate through message passing, as in the actor model. Processes in Erlang are not managed by the host operating system, but instead are managed by the Erlang environment itself, making them extremely light-weight in terms of memory consumption.

Erlang code is not structured into independent processes, rather an Erlang program consists of a collection of functions, each of which can act as a starting point in the creation of a new process. This therefore requires careful thought in regards to code organization on the programmers behalf to ensure that the runtime behaviour of each process can easily be discerned from the source.
Creation of a new process takes place through a call to the ‘spawn’ function, using the following format:

\[ \text{spawn(<module>, <function>, <arguments>)} \]

Each function has an associated module, similar to a package in Java, and when a process is spawned a set of initial arguments are given to the seed function. The return value of a call to spawn is a Process Identifier (PID) which can be used to send messages to the newly created process.

Messages can be sent between Erlang processes using the ! operator, as shown below:

\[ \text{<recipient PID> ! <message>} \]

Message sends in Erlang do not directly map to a function invocation at the receiver; To receive a message an Erlang process must instead make use of a special ‘receive’ construct. Such constructs consist of a number of patterns, each attempting to catch certain Erlang terms, and associated code blocks.

**Distributed programming**

Since Erlang originated from the domain of telecommunications, it is therefore only natural that it should have excellent support for distributed programming; This is indeed the case.

Once a process has been created it can be given a unique identifier through the use of the ‘register’ function. Processes operating in external systems may now communicate with such registered processes by sending messages to \{<registered process identifier>, <node identifier>\}, where <node identifier> consists of a node name (a node being a host environment for a collection of actors) and a hostname of the machine on which it resides.

Processes may be started on the current node, or alternatively they can be started directly on a remote node. To start a process on a remote node simply requires that an additional argument, giving the remote node identifier, be passed to the spawn function.

A deficiency of the distribution model present in Erlang is that it does not allow for the migration of processes between hosts. This is a feature present in several other concurrency oriented languages such as SALSA.

**Example**

The use of processes and distributed communication in Erlang will now be demonstrated through a simple echo service example.
The code (Listing 2.1) is initiated by typing “echo:start().” into an instance of the Erlang shell, ensuring that we have a additional node, “node1”, already running on the machine with hostname remotehost.

When the start function is initiated it begins by spawning a remote process on the node identified by node1@remotehost. This remote process is given the server function as a starting point and registered to the unique name ‘echo_server’. Upon creation, the server will print “Starting echo server” and then suspend inside the following receive block. After receiving a message of the format {echo, Text} the server will print the given text to the standard output and again re-enter the receive loop by means of calling the server(ready) function. If the message received is a terminate message, however, the server will call the built-in exit function.

The next line of the start function creates an additional process on the local node (as no node was explicitly given to spawn). This process is seeded with the client function, which sends the messages “hello” and “world” to the remote echo service before telling it to terminate. The argument {echo_server, node1@remotehost} given to the client function serves to identify the remote server process.
Erlang represents functional programming at its best, with all the features - such as good support for pattern matching - that functional programmers have become accustomed to. Its main strengths are its light-weight process model, which is naturally highly scalable, and its support for distribution. Erlang’s success in the telecom industry has proven the power of asynchronous message passing architectures.

As mentioned previously, it is however important that careful consideration is given in regards to structuring of code, if one would like to be able to easily determine the runtime behaviour of an application from looking at its source. The other main drawback of Erlang is its lack of support for mobility of processes between nodes; it does however offer the ability to ‘hot-swap’ code without halting the system, but this gives the developer the extra burden of explicitly passing around higher-order functions.

2.3.2 Scala

Scala is statically typed, pure object-oriented programming language developed at EPFL. Functional programming is additionally supported [48], with features such as higher-order functions, pattern matching, and currying, being made available to the programmer.

Scala code is compiled to run on the Java VM, with previous versions also inter-operating with the .NET CLR [29]. Scala shares much of its type system and control structures with those of Java and C# [36]. An advantage of the interoperability with Java is that the plethora of libraries/frameworks available to Java programmers may also be accessed via Scala.

Actors are not a core component of the Scala language, but instead can be made available by use of a library written by Phillip Haller [16]. The remainder of this overview shall focus on said library.

Haller’s library offers two actor implementations: a traditional thread-based approach, and Haller’s own event-based approach. Each approach has associated advantages and disadvantages, as will now be discussed.

Note: See Appendix A.1 for an introduction to Scala’s basic syntax.

Thread-based Actors

In this approach each actor is assigned a unique thread from the host environment [38]. When one would like to suspend or resume an actor, they need simply suspend or resume the respective thread.

Thread-based actors benefit from a relatively simple implementation, and through assigning each actor a unique thread, problems of system deadlock which might
otherwise arise when performing blocking operations are eliminated; actors do not wait on each other, and therefore all actors besides the blocking actor can continue operations as normal.

The drawback of this approach however is that assigning each actor its own native thread takes a big hit on system resources. Since each thread in Java is recognized as an independent process, this means that resources are consumed further still.

**Event-based Actors**

Using this second approach an actor is decoupled from any unique thread of control. When an actor is suspended – following a receive - a closure representing its remaining computation is saved; contrast with the previous approach in which the execution state of an actor is managed by its respective thread’s stack and program counter. A sleeping actor is then woken by “piggy-backing” the execution of the closure on the thread of an actor attempting to send a message to the sleeping actor [37].

Although it is possible to run a group of Event-based actors on a system with only a single thread, this has a number of shortcomings, most notably:

- If an executing actor decides to perform any sort of blocking operation – besides a receive – then the whole system grinds to a halt.
- Potential parallelism, such as that offered by multi-core processors, is lost.

To deal with the problems of the single-threaded approach a ‘thread pool’ is introduced. Each worker thread in the pool is capable of running a ‘task’, where a task represents the resumption of a previously suspended actor; Actors can again be detached from the assigned thread in the case that they perform an unsuccessful receive.

In the case that all threads are in use when a new task comes in, then a new thread will be created assuming all in-use threads are blocked; if a single worker thread is active (not blocked), however, the task will be queued as to prevent the creation of too many threads [37].

Since we do not assign each actor a native thread in this approach, but instead have a limited size thread pool, event-based actors therefore benefit from high scalability. In [38] Haller notes that when using a Java heap size of 512MB, the VM runs out of memory once 5500 threads have been created; 5500 is therefore a limit on the number of thread-based actors in such a system; with the event-base actor approach, however, it is possible to have up to 1.2 million simultaneously active actors using the same size heap.

Event-based actors in Scala have three main drawbacks as far as I can see:
The implementation is complicated by introducing new factors such as a thread pool, task objects, and a scheduler to assign such tasks to worker threads.

Event-based actors in Scala suffer from a restricted programming model. This stems from the fact that because Java does not support first-class continuations or explicit stack management, it is therefore not possible return from a receive call [37].

The scheduler in an event-based actor system is only called upon a send, or when an executing actor detaches from its worker thread [37]. Combine this with the scheduling policy mentioned previously (i.e. no new threads are created if an actor is still executing) and problems of reduced concurrency can incur. Take for example the following scenario:

An event-based actor system exists with two worker threads in its thread pool. Two actors, A and B, are created and assigned to these worker threads. Some-time during there execution A and B spawn two new Actors C and D and continue there own execution. Since A and B are still running, the execution of C and D will be queued in the task queue. Once C and D are queued, A and B then call blocking I/O operations. The system now grinds to a halt until these blocking operations are completed, even though C and D are ready and willing to execute.

**Unification**

Scala allows developers to take advantage of both approached by unifying thread-based actors and event-based actors in a single programming model [38]. The default ‘receive’ privative corresponds to a thread-based actor. An additional primitive ‘react’ can be used instead of receive if the programming would like to follow the event-based model.

**Evaluation**

Scala is a modern programming language with mounting support. Its ambitious aims to fuse both object-oriented programming and functional programming gives the developer increased power - through constructs such as pattern matching, but at the same time could seem confusing to those coming from a Java background, where such features do not exist.

Implementing the actor model as an external library, as opposed to part of the core language itself, was a wise decision; Scala provides good support for adding new language constructs through external libraries, and at the same time this means that a custom build of Scala is not required to exploit the Actor model - assuming it wouldn’t have otherwise been merged into the main repository.

The library itself is quite flexible in that it offers two implementations of the actor model, but a tough decision needs to be made as to whether increased
scalability should come at the cost of the potential for reduced parallelism and a more restricted programming model. The library offers support for distribution, however this does not appear to be well documented.

2.3.3 SALSA

SALSA [15] is an ‘actor-oriented’ programming language. The language inherently supports the Actor model while at the same time it attempts to retain the benefits of an object-oriented language [27].

After creating a SALSA application, the code is fed into a preprocessor which outputs an equivalent Java source file. The Java source code, which makes use of a special Actor library [52], can then be compiled to Java Bytecode and executed on the Java VM. As in the case of Scala, this makes existing Java libraries/frameworks available to the SALSA programmer.

SALSA does not have any explicit notion of specifying replacement behaviour of an actor through the ‘become’ keyword, as seen in [17]. Changes to behaviour are instead implicit and result from changes to an actor’s internal state; Internal state can be modified by giving new values to internal variables through assignments or local method invocations [52].

Messages in Scala are mapped directly into method invocation at the receiving actor; This is an instance of the ‘active object’ pattern [43]. Traditionally messages in actor languages are pattern matched at the receiver in a special ‘receive block’. The author believes that the active object approach is a lot more natural for object-oriented programmers and is therefore well placed in SALSA.

Note: See Appendix A.2 for an introduction to basic SALSA syntax.

Continuation Passing Style

Due to the one-way asynchronous nature of actor systems, coordinating interactions between multiple actors can be complicated. A trivial example of such coordination would be where some actor A sends a message to an actor B asking for some value to be computed and would then like the computed value forwarded to a third actor C.

To handle the issue of coordination, SALSA makes heavy use of continuations; a continuation being the code representing the remaining computation to be carried out once a message has been processed.

Three types of continuations are available in SALSA: token passing continuations, join blocks, and first-class continuations. These different flavours shall now be discussed.
**Token-Passing Continuations**  Token-passing continuations allow the programmer to specify chains of message sends, where the output of one link in the chain can be fed as input into the next link by means of a special ‘token’ keyword. Since message processing for a link in the chain must complete before its successor is invoked, this allows a partial ordering of message processing to be specified [27].

The following example demonstrates the usage of token-passing continuations in SALSA:

```plaintext
1  fileSystem <- read('graphic.jpg') @
2  imageProcessor <- transform(token) @
3  fileSystem <- write('transformed-graphic.jpg', token);
```

In the first line of the above code the current actor sends a ‘read’ message to the fileSystem actor, specifying that it would like to read the contents of ‘graphic.jpg’. Continuations are identified by a preceding ‘@’, with a ‘;’ delimiting the end of a ‘continuation chain’. The read data is now passed to an imageProcessor actor by using the ‘token’ keyword. Finally the transformed data, also specified by the token keyword, is written to the filesystem.

A subtlety with continuations in SALSA is where message sends specified in a continuation chain are actually executed. Although it may appear that the current actor executes all three message sends specified in the example, this is not the case. The current actor only initiates the first send in the chain (i.e. the read message) and then continues with its remaining computation following the ‘;’ delimiter. The second message send will be initiated by the fileSystem actor following processing of the read message, and similarly the final message send will be initiated by the imageProcessor actor after transforming the image data it was given.

To demonstrate behaviour of message sends in SALSA without the ordering imposed by using token-passing continuations we refer to a simple example given in [27]:

```plaintext
1  standardOutput <- print('Hello '); standardOutput <- print('World');
```

The behaviour of the above code is indeterministic as there is no guarantee that the first print message will get processed before the second. The data printed to the screen could therefore either be “Hello World”, or alternatively “WorldHello”.

**Join Continuations**  Join continuations are a mechanism by which a barrier can be placed on the completion of several parallel activities before computation can proceed. Results from the parallel activities are merged into a single object
array which is accessible in continuations following the join using the 'token' keyword mentioned previously.

Assuming an array containing three URLs, a search engine could distribute crawling of the URLs by using a join continuation and then store the results in the following continuation as follows:

```java
join {
    crawlerOne <- crawl(urls[0]);
    crawlerTwo <- crawl(urls[1]);
    crawlerThree <- crawl(urls[2]);
} @ searchIndex <- store(token)
```

**First-Class Continuations**  First-class continuations give the receiving actor explicit access to the current continuation through the keyword ‘currentComputation’. This explicit access allows the actor to delegate computation to a third party independently of the current message processing context [52].

The following examples, taken from [27], illustrate the effect of using the currentComputation keyword in the context of a simple Hello World application:

```java
.. void saySomething1() {
    standardOutput <- print("Hello ") @
    standardOutput <- print("World ") @
    currentContinuation;
}

saySomething1() @ standardOutput <- print("SALSA");
..  

This example is guaranteed to print “Hello World SALSA”.

```

```java
.. void saySomething2() {
    standardOutput <- print("Hello ") @
    standardOutput <- print("World ");
}

saySomething2() @ standardOutput <- print("SALSA");
..  

This example could result in either “Hello World SALSA” or “SALSAHello World ” being printed. The second combination is possible because as soon as the first print message has been sent in saySomething2() control returns to the caller (i.e. the last line of the code) and the message to print “SALSA” is sent without waiting.

17
Distributed Programming

Computing in SALSA is not restricted to a single machine, but instead can be spread across a highly distributed environment.

Distribution is possible through the use of ‘Universal Actors’. Universal Actors are capable of migrating between ‘theatres’ (a host environment for a group of actors) and may be referenced by actors in external theatres.

Referencing a Universal Actor is possible thanks to ‘Universal Actor Names’ (UANs). Each Universal Actor has a UAN which is unique throughout its lifetime. Once an actor’s UAN is known, a reference can be created through the use of the ‘getReferenceByName’ method offered by SALSA; the reference can then be used in a manner analogous to that of local references.

A Universal Actor can be migrated to a different theatre by sending it a ‘migrate’ message containing the location of the new theatre in the form of a ‘Universal Actor Location’ (UAL). The UAL consists of a messaging protocol, a hostname, and a port number.

Evaluation

SALSA represents the state-of-the-art in the current generation of actor languages. The language does a good job of bringing the actor model to the world of object-oriented programming; with actor definition, instantiation, modification and message passing following a style which will be intuitive to programmers with an object-oriented background.

Continuations provide for a powerful mechanism by which multiple actors can be coordinated. Determining where a particular piece of code will be executed at runtime can be somewhat confusing at first however - it is not always in the actor where the code was defined, and continuations are not a feature that will be familiar to most Java programmers.

SALSA has excellent support for both distribution and mobility of actors, thanks to the Universal Actor concept discussed previously. The approach for registering actor names and retrieving remote references should come naturally to anyone who has previously developed distributed applications - e.g. using Java RMI.

2.3.4 Stage

Stage is an actor-based language embedded in Python and was created by former Imperial College student John Ayes for his final year individual project [24].

Embedding the language in Python means that most of the constructs and features of Python are retained. There are however a few features which have been prohibited, these fall into the categories of: threading, first-class functions, and external interaction. The reasons for such prohibitions are that the features
are either redundant with the introduction of actors, or because they could leave actors in an unresponsive state/leave local resources in an inconsistent state [24].

Actor definition is performed by extending the built-in Actor class. Rather than defining behaviour inside a receive block, Stage has taken an active-object [43] based approach in which message sends map directly to method invocations at the receiving actor.

The following code shows the definition of a simple actor which multiplies two numbers together:

```python
1 class Multiplier(Actor):
2     def multiply(self, lhs, rhs):
3         return lhs * rhs
```

When a message is sent to an actor - which happens in a manner analogous to a method call, the result of such a send is a future. A future is a place-holder for a computation which will be filled when the result is available; taking such an approach allows the calling process to continue its own computation until the return value is actually needed, thus increasing potential concurrency. It is possible for a calling actor to explicitly wait for the computation to complete by using the ‘sync’ function, as shown:

```python
1 ...
2 result = multiplier.multiply(2, 2) #sends request
3 sync(result) #waits for result to return
4 ...
```

Actors in Stage each reside in a ‘theatre’, this is a concept similar to an Erlang node. Actors residing on different theatres may remotely communicate with each other - through a concept known as ‘networked singleton actors’, and it is also possible to implement a special type of ‘Mobile Actor’ that can move between different theatres.

In conclusion, Stage boasts a rich feature set and its usage should come naturally to developers who are already versed in Python. Unfortunately the project has not released any deliverables to the public yet, and there is no sign that this will happen in the future either.

### 2.3.5 Ruby: Erlectricity

Erlectricity [32] is an effort by Ruby developer Scott Fleckenstein to build a bi-directional communication bridge between Ruby and Erlang. The work is currently in very early phases, and there is at present no documentation provided for the project and only a limited number of examples available - two to be exact.
From the examples [5], it is evident that there is a Ruby implementation of Erlang’s ‘receive’ block and the ‘!’ operator for sending messages to Erlang processes. A ‘match’ method has been introduced for use in pattern matching incoming messages inside a receive block; since Ruby does not have atoms like Erlang, match instead uses Ruby symbols [45] as a replacement.

The library does not offer a ‘spawn’ primitive as in Erlang, but rather each Ruby program using the library represents a single Erlang process. There is therefore no concurrency benefits to gain from using the library - on the Ruby side at least.

Looking at the examples, the following code was given in a sample Erlang process:

```erlang
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

This seems to suggest that an Erlectricity program (here ‘gruff_provider.rb’) cannot be executed independently, but instead must be started from within the Erlang process with which it is to communicate. There is therefore a strong binding between the Erlang process which calls open_port and the Erlectricity process, and from my understanding this binding must remain fixed throughout the life of the Erlectricity process.

In summary, the library in its current form offers a nice local communication bridge between a Ruby application and a single Erlang process, but it is not helpful in terms of bringing a better concurrency model to Ruby, nor does it offer anything as far as migration or distributed programming are concerned.

### 2.3.6 Ruby: Omnibus Concurrency Library

The Omnibus concurrency library [10] attempts to fully bring the Actor model to the Ruby world. Like Erlectricity, it is currently in very early phases of development, with little documentation and only a few pieces of example code available.

The below is a piece of code found on a Ruby message board [1], posted by the library creator, MenTaLguY:

```
1
2
3
4
5
6
7
8
```

In summary, the library in its current form offers a nice local communication bridge between a Ruby application and a single Erlang process, but it is not helpful in terms of bringing a better concurrency model to Ruby, nor does it offer anything as far as migration or distributed programming are concerned.

### 2.3.6 Ruby: Omnibus Concurrency Library

The Omnibus concurrency library [10] attempts to fully bring the Actor model to the Ruby world. Like Erlectricity, it is currently in very early phases of development, with little documentation and only a few pieces of example code available.

The below is a piece of code found on a Ruby message board [1], posted by the library creator, MenTaLguY:

```
1
2
3
4
5
6
7
8
```

In summary, the library in its current form offers a nice local communication bridge between a Ruby application and a single Erlang process, but it is not helpful in terms of bringing a better concurrency model to Ruby, nor does it offer anything as far as migration or distributed programming are concerned.
greeting = "Hello"
loop do
  # guarded receive (does case-like matching via #===)
  Actor.receive do |f|
    f.when Greeting do |m|
      greeting = m.value.dup.freeze
    end

    # callback part 1
    f.when RequestGreeting do |m|
      m.reply_to << Greeting[greeting]
    end
  end
end

# sending a message to an actor
oracle << Greeting["Howdy"]

# getting a reference to the current actor
current = Actor.current

# callback part 2
oracle << RequestGreeting[current]
Actor.receive do |f|
  f.when Greeting do |m|
    puts "#{ m.value }, Bill!"
  end
end

From this code we can see that a Ruby class called 'Actor' has been introduced, and a corresponding spawn command is offered for the definition/creation of new actors. Like Erlang, messages are received inside a receive block, with a new method 'when' being used for pattern matching.

When the static spawn method is called it is given a Proc (code block) and the result is that we simultaneously define an Actor and create a single instance of such an Actor. Actors therefore follow the prototype style of programming, and if one would like to create multiple instances of an identical actor they must either duplicate the code, encapsulate it in a method, or call clone on a created actor. Considering that Ruby itself is not naturally a prototype style programming language, this decision therefore seems quite confusing if the library is going to be used for anything other than small on-the-fly actors (anonymous actors). The alternative would have been to let developers extend the Actor class when they want to define a new breed of actor.

The above code shows that a message can consist of a single Object only - in the example above a Struct instance. This appears to be a very cumbersome approach to message passing. Take for example an 'addition' actor which computes the result of adding two integers together; before this can happen, the developer must first define a new Struct to hold two integers, and whenever a message is to be sent to the actor an instance of the Struct must be instantiated with the two integers to be added.
The ‘when’ method provided allows incoming messages to be patterned matched based upon class - as in this example, or alternatively based upon the content of the struct object if using MenTaLguY’s ‘Case’ package [10]. Pattern matching is not familiar territory in the Ruby community, being more commonly found in functional programming languages, however it none the less proves a useful feature in this instance - given that only a single object can be passed.

Besides the features offered above, the Omnibus concurrency library also supports the use of Futures. A future in the Omnibus library is not a placeholder for the result of a message sent to an actor, as in Stage (2.3.4), but rather a future represents a standalone computation which can be started asynchronously and synchronized when its return value is needed. The below code, taken from [6], shows this:

```ruby
require 'concurrent/futures'
include Concurrent::Futures

f = Future.async { 3 * 3 } # the block runs in a new thread
f.inspect # => #<Thunk #<Thread:0xdeadcafe run>>
f + 1 # waits for thread to complete, then => 10
f.inspect # => 9
```

The Omnibus concurrency library shows that other Ruby developers see the potential benefits that the Actor model could bring to Ruby. It is unfortunate however that it mixes in concepts from the functional programming realm or concepts that are otherwise unfamiliar in the Ruby world. Additionally, it does not scale to multiple cores (all actors run in a single VM instance), and like Erlectricity it offers nothing in the way of distribution or migration.
Chapter 3

Specification

This chapter of the report outlines a number of high-level requirements for the AiR library. The aims for AiR were given in section 1.1.

3.1 Integration with Ruby

AiR provides a language for concurrency that can be used by Ruby programmers, i.e. an internal domain specific language [33]. Ruby offers a wealth of powerful features, and has a large number of third party libraries to perform a number of common tasks. It is therefore important that applications built with AiR are capable of taking advantage of this and should not be executed in isolation from the host language.

3.2 Message Passing Abstraction

Message passing between actors in AiR should be provided in a manner that is consistent with the object-oriented programming model, avoiding the usage of features typically found in the functional programming world such as pattern matching. When a request is made for some data this should be returned in a transparent manner, without requiring the definition of additional logic to handle the incoming response message.

3.3 Distribution

“The net is vast and infinite” - Ghost in the Shell (Film)
Applications built with AiR should not be bounded by the computational resources of a single machine. Instead, it should be possible to create large distributed actor systems across multiple computers that are connected to each other via a network. To permit this, it will be required that a method by which actors can initiate communication with each other be implemented. This process should be as pain-free as possible, and should take into account the possibility for the distributed system to dynamically change and evolve over its lifetime. Communication between distributed actors should be transparent and appear no different from communication between actors running in the same host environment.

### 3.4 Mobility

The implementation should provide the infrastructure needed to allow actors to migrate from one location to another. [35] highlights a number of key reasons why mobility is a good idea, amongst them are the support it offers for:

- Fault-tolerance
- Load balancing
- Maintenance and upgrades
- Service customization

### 3.5 Scalability

AiR applications should automatically scale to all available cores of systems on which they are resident. This is important as one of the key benefits of the actor model is that actors operate concurrently, and so not providing such scalability would seriously limit the efficacy of the language.

In addition to exploiting all available processing resources, it should also be possible to produce applications in which large numbers of actors are in operation at anyone point in time without this having a detrimental effect on performance.

### 3.6 Security

When operating in a networked environment it is important that mechanism are put in place to shield software systems from attacks by malicious users. AiR should provide facilities to support for this; for example, actors should be allowed to specify those components in the system that they trust and are willing to accept communications from.
Chapter 4

Implementation

This chapter will attempt to describe the various components present in a system created with AiR, and additionally will give insight into the implementation of important features such as naming and mobility. Rationale behind design choices made will also be given where appropriate.

When creating a piece of software there will be numerous criteria driving its design, and trade-offs will often have to be made based upon the relative importance of each. In AiR, the following factors impacted heavily on the design:

- Resulting ease of use for AiR developers
- Extensibility and modularization
- Scalability
- Performance
- Fault-tolerance
- Exploitation of parallel computing resources
- Ability to operate in a distributed, dynamically changing environment

For a detailed overview of AiR’s usage from a developers perspective please refer to Chapter 5 of the report.

4.1 Runtime Overview

An executing AiR application consists of a number of interacting actors, each of which resides in a host environment known as a ‘theatre’ - as is also the case
in SALSA (2.3.3) and STAGE (2.3.4). Actors forming a particular system may all exist within the context of a single theatre, or alternatively may communicate across theatre boundaries assuming a common nameserver is used by the theatres.

Figure 4.1 gives a top down look at an AiR system, and the roles of the individual components will subsequently be discussed in further depth.

4.2 Theatre

As mentioned, a theatre is the host environment in which a collection of actors executes. An actor system may consist of a single standalone theatre, or alternatively may consist of several interlinked theatres forming a distributed actor system. Theatres forming a distributed actor system all share a common name server; this is a directory service which can allow actors to associate a unique system-wide name with their mailing address, which can subsequently be used by actors in distinct theatres to initiate remote communication with each other. Naming is described further in section 4.5.

Supporting the open systems concept [42], theatres and actors can dynamically join and leave an AiR system. Theatres forming a distributed actor system need not all be resident on a single computer, as this would be rather limiting, but rather they are able to form systems across any computer network. Actors are vagrants that may travel between theatres as they see fit during the course of their lifetime; this is described further in section 4.4, where the implementation details of mobility are covered.
Theatres as described thus far are conceptual entities. In the actual implementation of AiR a theatre is composed of a concrete (i.e. mapped to a system process) ‘master theatre’ and a collection of concrete ‘slave theatres’. The rationale for this is discussed in section 4.2.3. A developer of an actor system built with AiR need not worry themselves with these details, but instead can think of a theatre as a single unit. Figure 4.2 illustrates the internal configuration of a theatre, with actors being represented by circles at the slave theatres.

From this point on the conceptual view of a theatre will be referred to as the ‘logical theatre’ or simply as a ‘theatre’.

### 4.2.1 Master Theatre

The master theatre primarily acts as a coordinator and has the following responsibilities:

- Execution of theatre script.
- Initialization of slave theatres.
- Distribution of actors.

**Execution of theatre script**

When a developer writes an object-oriented piece of software they must define a number of classes, and additionally they must define a piece of entry code
that creates initial objects and gets executed when the system is initialized. In Java this is a method with the signature ‘public static void main(String args[])’ defined in the class given to the Java VM, and in Ruby it is a standalone piece of code - transparently added into the top-level ‘main’ object - enclosed in an if statement that checks whether the current file is the one given to the interpreter. In AiR the situation is analogous; the developer defines a number of Actor classes, and they also define entry code to create the initial actors at the (logical) theatre. We refer to the entry code as the ‘theatre script’ and individual actor class definitions as ‘actor scripts’.

When a program that uses the AiR library is executed, it will be in the initial Ruby virtual machine instance that the master theatre resides. It will also be here that the theatre script, which in effect is nothing more than Ruby code that uses the AiR library, will be executed. An example theatre script is given in section 5.3.

**Initialization of slave theatres**

In AiR each slave theatre operates in its own virtual machine instance, as discussed further in 4.2.3. Since the master theatre is bound to the initial virtual machine instance it is therefore tasked with the job of starting the slave theatres, one (or more) per CPU core.

It was unfortunate to find that there was no platform independent way to do this in Ruby, and so a number of operating system specific libraries were made use of. Fortunately this was the only part of the code that required platform dependencies, making the task of providing versions for both Windows and Linux trivial. It should be noted that it is still possible to build actor systems that comprise of interacting actors running under both Windows and Linux.

The following code is taken from the Windows version of AiR and shows the library call that act to spawn the slave theatres:

```ruby
@cores.times do |i|
  Process.create :cwd => File.dirname(__FILE__) + "/../",
  :app_name => "ruby spawn_slave.rb slave_#{i} #{@uri} '
  '#{main_file}' #{@name_server_uri}"
end
```

Process.create tells the operating system to start a new process with the given command (:app_name), starting in a specific directory (:cwd). In this case the application is the Ruby virtual machine, which is fed a script that loads the slave instance (‘spawn_slave.rb’) along with a number of additional parameters for initialization and callback communication with the master theatre.
Distribution of actors

When a request is made for a new actor to be spawned, or for an existing actor to be migrated to the current logical theatre, it is the responsibility of the master theatre to choose a slave theatre to host the actor. Requests to spawn an actor may be found in the theatre’s script, and additionally inside an actor’s own script; for example, an actor that performs a scientific calculation may decide to split the calculation into a number of independent computations that could be distributed to newly created worker actors. An actor in a remote theatre is able to migrate to a different theatre given that the destination theatre has registered a name that can be looked up at the nameserver.

The distribution strategy employed in AiR is to uniformly distribute actors among the slave theatres. This is rather simplistic, however for the examples implemented it has proven satisfactory. A possible extension, not implemented due to time constraints, would be to allow developers to define their own custom (domain specific) distribution strategies that could be specified upon initialization of a theatre.

4.2.2 Slave Theatre

Slave theatres represent the environment under which actor instances are actually executed. To provide this, the slave must perform the following duties:

- Creation and destruction of actors
- Coordinating communication between actors
- Provide support for migration (4.4)

Creation and destruction of actors

Upon receiving a request to spawn an actor from a master theatre a slave theatre must instantiate the requested actor, making sure to allocate the actor a unique ID amongst all actors at the slave. This unique ID, along with the address of the theatre itself will then serve as the actors mailing address. The slave maintains a registry of all actors that it hosts, indexed by ID - for later interaction. Once started, an actor runs autonomously in its own green thread, as discussed in 4.3.1, interacting with the slave theatre when it needs services such as communication with other actors.

When an actor is destroyed, for example through a call to the ‘kill’ method (5.10.2), or after migration to a new theatre, the slave theatre will remove any references to the actor and ensure that any associated threads are destroyed. The memory used by the actor will then be automatically reclaimed by the Ruby garbage collector. A log is kept of of such events, for example to allow forwarding of messages to migrated actors.
Coordinating communication Between Actors

When one actor wishes to send a message to another actor it is the responsibility of the associated slave theatre(s) to ensure that it is delivered to the destination actor’s mailbox. After a message send request, generated using a proxy as discussed in 4.3.2, has been received from an actor by the slave there are a number of alternative courses of action that might be taken. For now we shall consider the case of a one way communication (i.e. when a return value is not required).

One-way Communication

Should the communicating actors reside on the same slave theatre this is a simple case of using the destination ID included in the message to retrieve the corresponding actor’s reference from the registry and to then deposit the message in the actor’s mailbox, by calling the actor’s ‘deposit_message’ method - something all actor instances posses. It should be noted that any arguments passed in messages between actors are first marshalled (flattened into byte sequence) and then reconstructed on receipt as to prevent shared state, which is prohibited by the actor model.

If the destination actor resides in a separate slave theatre then a custom communication library is used to pass the message between the two theatres, after which point a similar process is performed. The first attempt at making a communication library was based upon the DRb (Distributed Ruby) RMI library [8] which is available for Ruby. It was soon discovered, however, that due to the synchronous nature of DRb this would have a huge impact on performance; if the destination theatre’s communication thread suspends before a message has been deposited, then this would mean that the source theatre’s communication thread must also suspend.

To improve performance, the decision was made to write a new asynchronous RMI library from scratch, using the TCP socket classes provided by Ruby, that would return immediately after a remote call is made. Although no official benchmarks were taken, from running a number of sample AiR systems, such as a simple Ping Pong application, the resulting speed-up was evident. Allowing methods to be invoked on a remote theatre, rather than having the receiver pattern match raw messages received over a socket, allowed for a smoother transition as communication protocols between collaborating theatres matured during AiR’s evolution; it will also mean that any extensions or modifications to AiR in the future will be as pain free as possible from the perspective of communication.
Two-way Communication

Two-way communication in AiR is performed in a similar manner to SALSA and Stage, whereby a placeholder object is returned to the sender immediately upon dispatch of a request. This approach draws heavily on the related concept of a ‘future’; a future is an actor that represents the value of a parallel computation, being created when appropriate, thus freeing the creator to do other work, and being synchronized if/when the value is required [44][39].

In AiR we refer to the placeholders mentioned as ‘futures’, however in reality they are not real actors, but rather they are objects shared between the host theatre and the actor that performed the message send. The future object is created by the theatre upon receipt of the message send request, and it is then stored along with an ID that is included in the message that eventually ends up at the destination actors mailbox. When a message has been processed by an actor, a check will be performed to see if the message requires a result (i.e. a future was issued), and if so the result will be returned to the originating theatre accordingly. Once back at the theatre the future will be filled, thus making the result available to the actor. The future object will deal with any additional work that is required such as awaking the actor’s thread if it went to sleep after attempting to synchronize the future.

4.2.3 Theatre Design Choices

One of the motivations for developing the Actor model was the prospect of highly parallel computation devices [31]. With the mass adoption of multi-core and multi-processor machines it is therefore vital that AiR should fully exploit the potential parallelism afforded.

The simplest way to achieve maximum parallelism between actors, and to thus exploit the potential of these new chips, is to assign each actor a native system thread to process messages in its message queue. Due to the share-nothing approach of the Actor model, these threads can then be executed completely independently from each other, making use of any available cores.

As previously mentioned in 2.2, Ruby makes use of green threads. While native threads are scheduled by the underlying operating system, green threads are scheduled by the virtual machine itself; the Ruby virtual machine is only ever resident on a single core at any given time, and as such any parallelism afforded by a multi-core system is not taken advantage of by default. It was for this reason that the approached detailed above, in which master and slave theatres each execute in their own virtual machine instance, was taken. This gives an actor system the potential to execute in true parallelism on all cores of the host system, instead of limiting it to run under single core inside a single virtual machine instance.
4.3 Actors

The fundamental unit of abstraction in the Actor model is that of the actor. Like objects, actors consist of state and behaviour, however unlike objects they operate concurrently by default and communicate asynchronously rather than synchronously. Associated with each actor is an addressable mailbox into which messages can be sent for processing by the actor. Please refer to section 2.1 for a complete overview of the model.

When implementing actors in AiR, a library based on the object-oriented programming language Ruby, certain choices were immediately clear; for example, how does one represent the behaviour and state of an actor: use classes and objects. Other choices, in particular how to best achieve concurrent execution of actors in a scalable manner and selection of an appropriate abstraction for asynchronous message passing, were not as easy to make. This section of the report attempts to explain and justify the decisions made in AiR with respect to these questions.

4.3.1 Achieving Concurrent Execution

Actors in the pure actor model achieve their concurrent execution by the fact that they each poses a unique thread of control for processing messages in their mailbox. Mapping such a (conceptual) thread of control in a concrete implementation of the model provides two main alternatives, namely: threads and processes. The semantics of threads and processes in Ruby, and the relative merits of each, were discussed in 2.2.

An alternative approach to assigning each actor its own unique thread of control is to make use of a ‘thread pool’. Using this strategy a collection of idle threads is maintained (the pool), and when it is observed that an actor has a non-empty mailbox a thread from the pool may be assigned to process one or more of these messages. After finishing processing the message(s) the thread will be re-added to the pool.

Thread pools have the advantage that they are scalable in regards to the fact that actors need not have a unique thread to themselves any more, thus reducing memory consumption. There may also be a reduction in context switching, depending on the scheduling policy used. Coupled with the advantages just mentioned are a number of deficiencies; the first is that in situations where there are more actors with non-empty queues than there are idle threads a trade-off must be made as to whether the total number of worker threads should be increased, or whether actors should simply be left to wait - potentially for a long time; a second related disadvantage is that once a thread has been assigned to an actor, an actor has no way of releasing control of the thread until the message has been completely processed, at least without requiring extra work from the developer of the actor system. In a situation where blocking operations are
used from within an actor this could present a scenario in which threads that are doing no work are not available to the pool to process messages as they still have further work to do once the blocking operation is complete.

After weighing up the relative advantages and disadvantages of the different options discussed here and in the introduction to Ruby section, the decision was made that actors in AiR will each be assigned their own unique green thread. Due to the fact that green threads are far lighter-weight than native threads and full blown system processes, a thread pool was not seen to be necessary for scalability and would have introduced unwanted side-effects. In section 4.2.3 it was mentioned that slave theatres each operate in a distinct system process, this therefore means that actors in separate slaves can operate in true parallelism should the system provide for this. Thanks to the use of green threads, actors within a single slave theatre can operate with simulated parallelism and without fear of blocking other actors thanks to the to the fair scheduling policies of the Ruby virtual machine.

### 4.3.2 Message Passing Abstraction

All communication between actors takes place asynchronously, with control being returned to the requesting actor immediately after a message is made. This is in contrast to traditional synchronous communication model of object-oriented programming in which the same thread of control used by the requester is also used to process the request, hence prohibiting the requester from making progress until a result has been computed.

The common approach, as employed by Erlang for example, to asynchronous message passing is to introduce two new operations: send and receive. Send, as the name would suggest, is used for dispatching messages asynchronously, and is given as input a message - such as ‘likes(alice,bob)’ - and a destination. Receive is a pattern matching operation that is applied to all incoming messages until a match is found, after which point the clause associated with the matching pattern is executed. Listing 2.1 shows a simple echo server written in Erlang using this approach (Note: ‘!’ is Erlang’s send operator).

This approach, in my opinion, would appear quite unnatural to most Ruby programmers who are used to traditional synchronous method call/return semantics and who are not acclimatized with pattern matching - something usually restricted to the realm of functional programming. It was for these reasons that an alternative abstraction was sought, with the final decision being made to use the ‘Active Object’ pattern. The active object pattern provides asynchronous message passing in a manner more consistent with the object oriented programming model, and a further discussion of the pattern and its implementation in AiR will now be given.
Active Object Pattern

In the Active Object pattern, a message send event happens in a manner analogous to a standard method call. On the receiving side, a message need not be pattern matched inside a receive statement, but instead it is automatically mapped into a method invocation. The question is then, what is the difference between the active object pattern and regular object oriented programming? The answer is that method invocations are decoupled from method executions [active-object]. To achieve this decoupling, a number of new components must be added to the traditional object oriented model: an activation queue (mailbox) associated with each object, a scheduler, and the concept of a client side proxy. The latter two components shall now be explored further.

Client-side Proxy

To send a message in the Actor model it is first required that the sending actor has a handle on its target in the form of a mailbox address. In the Active Object pattern the analogy is that the sender requires a proxy object for communication with the destination active object; obtaining a proxy can be achieved in the same manner as obtaining an actor’s address (2.1).

A proxy is an object that provides an interface conformant with the target object (‘servant’) for which it is a proxy. Method invocations on the proxy’s interface are automatically converted into messages, containing amongst other things the method name and arguments, which are then dispatched to the target’s mailbox.

The implementation of a proxies in AiR relied heavily on Ruby’s strong meta programming support which provides facilities to add, remove, and modify methods at runtime, amongst other things. Quite possibly the most beneficial feature made use of was the ‘method_missing’ method [26] which allows an object to intercept calls to methods that do not exist. This is important as the interfaces of objects in Ruby can dynamically evolve over time, and as such a proxy should not be bounded by the initial offering of methods that a servant provides; indeed, in AiR an actor can even include such a method_missing method itself to intercept messages sent to it that ask for undefined methods to be invoked. The technique was additionally used to provide a fluent interface for customizing outgoing messages (5.4).

The following example shows a simple usage of the method missing technique in which the name of the undefined method called is printed out along with its arguments:

```ruby
1 def method_missing(method, *args, &blk)
  2 puts "Call to undefined method intercepted..."
  3 puts "Method name: #{method}"
  4 args.each_index do |i|
```
puts "Argument #i: #{args[i]}"
end
end

In the case that a return value is required, the calling actor will have a future object (4.2.2) returned as the result of a method invocation on the proxy.

Scheduler

Associated with each servant is a scheduler. Schedulers each have their own unique thread of control - in line with the fact that actors have a unique thread of control, and they are tasked with the job of: inspecting the activation queue, removing messages, and invoking/executing the corresponding method with the arguments encoded in the message. Once the method has fully executed, the scheduler ensures that the result, when required, is returned to the requester. In AiR, the result and original message are passed on to the host theatre which then uses the message to determine the host theatre of the initiating actor, after which point the result will be packaged up and transferred to said theatre for processing.

Example Scenario

Figure 4.3 shows visually the interactions that take place when using a proxy object to send a message to an actor and retrieve a response via a future object. Three different phases of interaction are shown, with the number of '*' symbols attached to labels being used to distinguish these.

In the scenario there are two actors, labelled ‘Source Actor’ and ‘Destination Actor’, each residing on different theatres. For conciseness these will be referred to as ‘actor A’ and ‘actor B’. Actor A has a proxy for actor B, accessible via the local variable ‘dest’. At some point during the execution of A’s script the statement ‘res = dest.calculate(args)’ is reached. This represents the fact that A would like to invoke B’s calculate method, passing the value ‘args’ as a parameter, and having the result returned and stored in the variable ‘res’. When the calculate method is invoked on the proxy object this results in the transparent generation of the corresponding message to deposit in actor B’s mailbox. To transfer the message from the proxy to actor B’s mailbox requires that it first be passed to the theatre hosting actor A, which is also home to the proxy object, after which point it is forwarded to the theatre hosting actor B. The destination theatre retrieves B’s reference from its registry and deposits the message in its mailbox. After forwarding the message, the theatre that hosts actor A generates a future object as a placeholder for the result of the calculate invocation, and this is returned to A and stored in local variable ‘res’.

At some future time point, actor B’s scheduler will eventually extract the message from B’s mailbox, and to process this it will invoke the real calculate
Figure 4.3: Active Object approach used in AiR
method provided by actor B. The result of this invocation is then passed onto actor A’s host theatre, which looks up the corresponding future object and populates it with the real result.

Either before or after the message was processed, actor A executes the statement ‘puts res’. This says that it would like to print out the result of the calculate message that was sent. In the case that the result has already been returned, the future object ‘res’ will transparently be replaced by the real result (see 5.5) which will then be printed to the standard output. If the result has not yet been made available to the future then this will result in actor A being put to sleep until the result arrives, after which it will be woken up and the result will be printed.

4.4 Mobility

A key requirement that formed part of AiR’s specification (chapter 3) was that it must provide actors with the ability to migrate from one location (host environment) to another. Migration of an actor requires that the following items be transferred from one theatre to another:

- The code which defines the actor’s behaviour.
- The actor’s internal state (runtime data values)
- The current execution state of an actor [optional]

When the third item - the actor’s current execution state - is transferred this is known as ’strong mobility’ [35]. Strong mobility allows execution of the actor at the destination theatre to resume from the point at which it left its originating theatre.

The alternative is to use ‘weak mobility’. Using this approach the code and state of an actor is transferred together with initialisation data that will allow it to resume execution. The initialisation data could for example be a method that is called when the actor has successfully migrated.

One of the main argument in favour of strong mobility is that it allows long running operations to continue uninterrupted without any data being lost from their current execution phase [53]. Take for example the case of an actor performing a computationally intensive calculation, if the call to migrate is made when the actor has already made significant progress with the calculation this would clearly be a waste in regards to work already done. An additional advantage is that the developer is not burdened with the need to additionally produce initialisation code to resume an actor that has migrated.

The argument in favour of weak mobility is that if an actor is location sensitive (i.e. it relies on local services/resources) then it should be made explicitly aware
that it has been moved so as to re-establish any required relationships in the context of its new environment. An example of such an actor where this is true would be a diagnostic actor that continually checks the state of the local machine - CPU load, memory used, etc. - on which it is currently resident.

The initial design choice in AiR was that hybrid strong mobility approach should be used in which an optional initialisation method is called upon resumption at the destination theatre, after which execution would continue from the position it left off. This would allow mobility in AiR to reap the benefits of both strong and weak mobility.

After starting the implementation it soon became apparent that it would not be feasible to support strong mobility as Ruby does not allow threads or stacks (i.e. the execution state) to be marshalled. For this reason it was decided that a weak mobility approach must instead be taken.

Ruby does not support the marshalling of class objects, and as such a more low level approach was needed for transferring an actor's code. In the current implementation, whenever an actor that has expressed a willingness to migrate is spawned at a theatre its associated source files will be searched, using regular expressions, to locate the corresponding class definition which is then cached by the host theatre. Actor class definitions are described in section 5.2.1. If a request is made to migrate the actor its definition will be located and transferred along with the actor's local state to the target theatre. Upon arrival the actor's class definition will be evaluated, after which an instance will be created with an identical local state to that transferred, and finally an initialisation method 'migration_complete' will be invoked.

Once an actor has finished migrating any messages intended for it that are sent to its original host theatre will be forwarded to its new home, and at the same time the theatre hosting the actor with an out-dated reference will be informed of the new location. Should the actor have registered a system wide name, the nameserver will be updated with its new address.

4.5 Naming

It was previous mentioned that an actor's mailbox address consists of a unique ID and theatre address pair. These values are dynamic in nature and may change with different instantiations of the system, or as the system evolves over time and actors migrate between theatres. Take for example the case of an actor system which contains a chat server, if the host on which the actor system executes is changed then so will the address that identifies the theatre, which in turn will change the actor's own address.

To cater for such dynamically changing systems, AiR introduces the ability for actors and theatres to register a unique system-wide name for themselves with a nameserver. This provides a static way for an actor to be contacted; a
lookup request is made to the nameserver giving the fixed system wide name of the actor, and then the actor’s current physical address is returned. This is a similar approach to the one used in SALSA (2.3.3). Figure 4.4 illustrates the registration and subsequent lookup of a chat server actor with a nameserver - note: in reality the calls go via the host theatres of the actors.

When an actor is looked up by name the physical address returned in cached and wrapped inside a proxy object for subsequent communication with the actor. An alternative would have been to look up the address every time a message is to be sent, this however would have had a large impact upon performance. The only time there could be complications with the strategy taken would be if an actor that was looked up migrates to a new theatre, AiR’s approach for dealing with this was discussed in 4.4.

4.6 Summary

This chapter has detailed AiR’s runtime architecture and implementation details. Actors are given their concurrency through the use of green threads, and can be communicated with thanks to the use of proxy objects.

Besides actors themselves, the most important concept discussed was that of a theatre. Theatres support the execution of actors by providing them with a number of services, such as the ability to communicate with other actors in the system. The most notable decision in regards to the design of theatres was to use a multiple virtual machine approach, splitting a theatre into a single master
and several slaves; this was done to overcome limitations inherent to Ruby, thus allowing AiR applications to automatically exploit all available cores on any system on which they are resident.

AiR implements weak mobility. Originally it had been intended that strong mobility also be available, however due to the fact that execution state cannot be transferred in Ruby this was abandoned. A custom approach was implemented to support for the transfer of code between theatres.

Name servers can be made use of in the case that actors operating in distinct theatres need to initiate communication with each other. This approach is resilient in the face of changes to an actor’s actual address – which can vary between different instantiations of the system, or if an actor migrates between theatres.
Chapter 5

Language

The AiR library provides Ruby developers with an actor-based embedded domain specific language for writing concurrent applications.

In this chapter the language will be discussed in depth, and combined with the sample applications that come with the AiR distribution, developers should have everything they need to start writing applications using the library.

5.1 Getting started

To begin writing AiR applications it is first required that the AiR library be imported into a Ruby application. This can be be achieved with a single line of code, as follows:

```ruby
require File.dirname(__FILE__) + "/lib/air.rb"
```

The explicit “File.dirname(__FILE__)”, which makes the include use an absolute path, is required as relative paths change when background scripts – such as those to spawn slave theatres – are executed.

An application using the AiR library is executed in the same manner as a standard Ruby application; the main file is given to the Ruby interpreter as an argument, e.g. ‘ruby my_actor_application.rb’.

5.2 Actors

5.2.1 Behaviour definition

AiR provides developers with the ability to define and instantiate serialized actors (i.e. those with mutable state). Definition of an actor in AiR is performed
via the creation of a new class which extends the included ‘Actor’ class. This
definition is known as an ‘actor script’.

Listing 5.1 shows an example actor script:

```ruby
Listing 5.1: Product Catalogue Actor Script

class ProductCatalogue < Actor
  def init(products={})
    @products = products
  end

  def query_price(product_name)
    return @products[product_name][:price]
  end

  def add_product(name, price, quantity)
    @products[name] = {:price => price, :quantity => quantity}
  end
end
```

In the example an actor representing a product catalogue has been defined. The
actor’s behaviour is defined via standard Ruby methods. Each message received
by an actor will request that a certain method be invoked upon its removal from
the actor’s mailbox. Return values will be transparently delivered to the actor
that sent the message. Besides calls into AiR library methods, method bodies
can contain any standard Ruby code.

`init` is a special method called upon the creation of a new actor. It differs from
a normal constructor as it is called asynchronously.

### 5.2.2 Instantiation

After an actor’s behaviour has been defined, it is possible to create actor in-
stances through calling the `spawn` method on the corresponding actor class. For
example:

```ruby
laptop_cat = ProductCatalogue.spawn {:sony_laptop => {
  :price => 999,
  :quantity => 10 }}
```

This call will create a new catalogue actor with an initial database that contains
details of a Sony laptop. The result returned from the spawn call is a proxy
object. A proxy can be used to transparently send messages to an actor through
standard method invocations, see section 5.4 for more details.

The approach discussed above, in which actors are defined through standard
classes and communicated with via proxies, is referred to as the Active Object
pattern. It is discussed thoroughly in 4.3.2.
5.3 Theatres

When an AiR application is executed a theatre will be started automatically. The initial behaviour of the application, such as the creation of core actor instances, is defined through a ‘theatre script’. A theatre script is nothing more than a Ruby code block that is passed to a special method Air.start. Listing 5.2 gives an example theatre script:

```
Listing 5.2: Sample Theatre Script

Air.start do
  laptop_cat = ProductCatalogue.spawn (:sony_laptop => {
    :price => 999,
    :quantity => 10 })
  monitor_cat = ProductCatalogue.spawn (:syncmaster => {
    :price => 150,
    :quantity => 5 })
  online_store = OnlineStore.spawn([laptop_cat, monitor_cat])
end
```

In this example two instances of the product catalogue actor defined previously are spawned and then passed to an actor representing an online store. This (hypothetical) store actor may then for example take requests from a browser user and query the product catalogue actors to retrieve their contained data before returning it to the user as an HTML document.

5.4 Message passing

Actors communicate through asynchronous message passing, and as already mentioned, in AiR messages are sent transparently by invoking methods on proxy objects. Like regular objects, proxies can be (copied and) passed between actors, and can also be retrieved using a lookup call to a nameserver, as discussed in section 5.8.

AiR offers a number of alternative types of messages that can be sent. The default style of message passing is that of two-way message exchanges, and modifications can be made to this by using a fluent interface [34], as shall be detailed in this section.

It should be noted that messages may also be sent from a theatre script to an actor. The only limitation with this approach is that method callbacks are not possible, return values however are perfectly acceptable.

5.4.1 Two-way messaging

Traditionally, languages which offer asynchronous messaging do not allow return values to be sent directly; instead results must be sent to known a priori result
handler methods. In AiR, the result of a two-way message send is a result placeholder (a ‘future’) that can be used to access the actual result when it becomes available.

Given the proxy object ‘laptop_cat’, a two-way message asking for the method \texttt{query\_price} to be executed at the receiving actor and a result to be returned can be sent as follows:

\begin{verbatim}
1 laptop_price = laptop_cat.query_price(:sony_laptop)
\end{verbatim}

Transparency the laptop_cat proxy will generate a message representing the fact that the query\_price method should be invoked, and then have it sent to the laptop catalogue actor. The result of the invocation on the proxy is a future object that is returned immediately and can be used to retrieve the price returned from the query if/when it is actually needed. The usage of future objects is discussed in section 5.5.

\subsection{One-way messaging}

In the case that a return value is not needed, such as when a void method – one that returns nil - is called, it is more efficient to send a one-way message. This lets the receiver know that a return message need not be sent, and thus bandwidth is saved.

The following example shows how to send a one way message that asks for a new product to be added to the laptop catalogue actor:

\begin{verbatim}
1 laptop_cat.>>.add_product(:macbook_air, 2000, 50)
\end{verbatim}

The only modification to the two-way message passing style is that ‘\texttt{>}’ has been added between the proxy object and the add\_product call. Alternatively ‘\texttt{>}’ may be replaced by ‘\texttt{one\_way}’ if this is deemed to be more understandable.

\subsection{Priority message passing}

Sometimes an actor may wish to ensure that a message it sends is processed as soon as possible at the receiving actor. An example where this would be appropriate would be a load balancing actor that would like to find out the utilization of other actors in the system so that they can be migrated to more appropriate hosts if needed.

Internally this is achieved by having two mailboxes for each actor: a standard mailbox, and a priority mailbox from which all messages are processed before any message in the standard mailbox is inspected. If a standard message is
already being processed when the priority message arrives this will first be left to completion.

In the following example an actor representing an http handler is sent a priority message to find out details about its utilization in the last sixty seconds:

```java
result = handler.*.utilization(60)
```

As with one way messaging, only a small change needs to be made to make a message into a priority message. This time a call must be made to `*` before the final `utilization` method invocation is made. Alternatively a call to `priority` may be made.

### 5.4.4 Composite message configuration

In addition to being able to make a single modification to a message, such as to say it is a priority message, it is possible to chain such modifications together; one-way priority messages can be specified. This is done by nesting the relevant calls between the proxy and the final method invocation, for example:

```java
handler.*.terminate
```

In the above code a one-way priority message is sent to the http handler actor telling it to execute a method called terminate. The ordering of nested calls does not matter, `*:>` is equally valid.

### 5.5 Futures

#### 5.5.1 Synchronization

When an actor decides to retrieve the value that a future object is a placeholder for this process is called synchronization. In the case that the return message has already been received by the current actor then synchronization will happen instantaneously, else the actor trying to use the future will block until the return message has been received.

From a developers point of view the future object can be treated as if it were the result itself, with synchronization happening transparently when a method is called upon the future. Alternatively the synchronization can be made explicit through a `sync` call on the future, resulting in the actual result object being returned.

The following example demonstrates the use of futures in AiR:
monitor_price = monitor_cat.query_price(:syncmaster)

.. code which doesn’t depend on monitor price ..

puts "Monitor price: #{monitor_price}"

The call to query_price returns a future object immediately, after which computation proceeds as normal until the code listed on line 3 is reached. When an object is printed an implicit call is made to its to_s method, and this results in the synchronization of the monitor_price future. Once the future has been synchronized the method invocation that was made on it will be forwarded to the actual result object, which in this case would be a Fixnum object representing the number 150.

5.5.2 Status Check

Sometimes it may be desirable to check whether or not a result has already been returned, as opposed to blocking the program until the result is available. This could for example be used to provide a richer user experience by giving visual feedback when a result is not yet ready. The available property of a future is a boolean value that represents whether or not the result has been returned yet, as shown below:

if monitor_price.available
  #print out order
else
  #update loading bar and other GUI elements
end

5.5.3 Selective Wait

Inspired by the programming language Ada [2], AiR provides the ability to selectively wait on a group of future objects until one or more of them is ready to be used immediately. This is useful in scenarios where an actor does not care which specific result is available, but simply wants to process whatever it can. An example of this is a communication manager actor that can receive data from external systems through a number of socket actors; as soon as data has been sent over a socket it should be processed immediately, regardless of whether or not anything has been sent over the other sockets.

Listing 5.3 gives a code snipped that might be found in such a communication manager:

Listing 5.3: Selective Wait

socket_data = FutureGroup.new
The example starts with the creation of a `FutureGroup` object. From a developers perspective this can be treated as if it were an array - although its actual implementation is somewhat different. Futures are now added to the future group, each representing 1000 bytes of data received over the sockets in presupposed to exist array `@sockets`. Finally a call is made to `selective_wait` on the future group. The `:all` flag says that results for all of the `get_next` calls should be given as arguments to the attached block as and when they arrive. The alternative to this is to use the `:one` flag to specify that only the first result that arrives should be given to the block.

5.6 Message Chains

In the actor model each actor is an autonomous entity, however it will often be the case that for a task to be completed several actors must work together. In such a situation one or more of the actors must act to coordinate the communication. The simplest way to perform this coordination would be for the coordinator to make a request to the first actor, synchronize the result, and then pass this on to the next actor, synchronize the result of this second request, and so on. This would be quite inefficient as all messages must go through the coordinator before the task could make progress, and it would no doubt block the coordinating actor’s thread much of the time.

The solution offered in AiR follows the token-passing continuation approach of SALSA (2.3.3). Under this scheme it is possible to specify a chain of messages that will be processed in order, with return values from each message represented by a token that can be passed as an argument to the next message in the chain.

5.6.1 Specification of a chain

An example will now be presented which will act to demonstrate how a message chain can be specified. The application is that of a web forum, and the system comprises of the following actors:

- controller_actor: receives requests from a browser user.
- user_database: contains details of users on the web forum
- post_database: contains all postings made on the forum
• view_actor: the actor responsible for sending an HTML response

A request now comes into the system, being intercepted by controller_actor, to view all posts by user ‘bob’. The controller could fulfil this request by creating a message chain as follows:

```
user_database.get_id!(‘bob’) >
post_database.get_posts_by_id!(token) >
view_actor.render_list!(socket, token).end
```

This chain consists of three messages: the first retrieves the ID of the user bob; the second uses the ID retrieved to find all posts by bob; and finally the third sends the posts to an actor for rendering as an HTML document that can be returned to the user over the given socket. A number of new pieces of syntax have been introduced in this example:

• ! - this specifies that method being invoked on the proxy forms part of a chain.
• > - places an order on items in the chain
• token – represents the return value from the previous message processed in the chain.
• .end – delimits the end of the chain

Once a message has been processed the result is sent directly to the next actor in the chain. This is more efficient in terms of number of messages sent than an approach where a central coordinator is used, it does however have the disadvantage that the remainder of the chain, and any objects passed as parameters, must be transferred to the next actor along with the result. Avoiding the inclusion of large data items as parameters when creating the chain would therefore be advisable.

In the example above, the view actor sends the result to the browser user and the task is completed. In some cases the actor that created the chain may want the result of the final message processed to be returned. This can be achieved by replacing .end with .return, which results in a future being returned as in the following example:

```
posts = user_database.get_id!(‘bob’) >
    post_database.get_posts_by_id!(token).return
```

```
.. operations not dependent on posts ..
```

```
posts.each do |post|
    puts “ID: #{post.id}”
```
5.6.2 Further uses of message chains

The primary role of message chains, as mentioned, is to coordinate messages sent between actors collaborating on a task. Message chains can, however, also be used for a number of auxiliary purposes, as shall be discussed now.

Ordering of messages

In the pure actor model there is no guarantee that if two messages are sent from one actor to another that they will be processed in the order sent. This also has practical implications as it means that additional processing is not needed upon receipt of a message to check that its predecessors have already arrived. Often times imposing an order could additionally restrict parallelism if the order in which messages are processed is not actually important - for example, when adding two distinct items to a database.

AiR does not by default require that messages sent be processed in order, if however this is a requirement then a message chain can be used to impose such an order. An example is given below in which messages are sent to an actor with the requirement that 'method_one' is invoked before 'method_two':

```ruby
actor_one.method_one!(some_data) > actor_one.method_two!(some_other_data).end
```

Should a return value be required from each ordered message, then message chains are not necessary and the synchronization of futures can be used to achieve the required ordering; if a message to invoke method_two is not sent until a result has been returned from the invocation of method_one, clearly method_one is processed before method_two.

Callback methods

It is not always desirable to have the result of some computation returned via a future, but instead it may be preferred that the result is sent to a special handler method. Message chains can be used to specify a callback method without actually having to explicitly code the callback into the behaviour of the actor that sends the result; the actor simply returns as normal, but transparently the result will be delivered to the handler method.

For example, suppose that an actor called ‘fibonacci’ can ‘calculate’ a particular Fibonacci number, and that the current actor would like the result returned to
its handler method ‘process_fibonacci’. This can be achieved as follows:

```erlang
1  fibonacci.calculate!(n) >
2  actor_self.process_fibonacci!(token).end
```

The current actor is now free to start processing other messages while it waits for the callback to ‘process_fibonacci’. `actor_self` is a method provided by AiR and allows an actor to retrieve a proxy to itself.

### 5.7 Dynamic Interfaces

In a concurrent application the messages that can be processed by an object vary over time depending on the current state of the object [47]. In a synchronous language like Java for example this would usually be dealt with by having the threads accessing an object continually check a condition to see if they can proceed, and if not to put themselves to sleep.

Actor systems must also deal with this problem. Take for example an actor which represents a buffer into which data items can be added (‘put’) and retrieved (‘get’). Without the use of synchronization constraints to specify when a certain message can be processed, if a ‘get’ message arrived at an empty buffer then the actor would have no choice but to return an error message.

#### 5.7.1 Receive Blocks

The solution to this problem in Erlang is to allow the specification of receive blocks (2.3.1). Such blocks list all the messages that can currently be accepted by a process, with all others being left in the process’s mailbox until a later point in time. AiR also provides this ability, however instead of pattern matching messages in the receive block, anonymous methods are defined which will automatically be invoked upon receipt of an appropriate message.

**Basic Usage**

The following code shows how an unbounded buffer actor could be defined in AiR with the use of a receive block:

```erlang
1  class UnboundedBuffer < Actor
2    def put(o)
3      receive do
4        get! { release o }
5      end
6    end
7  end
```
At the top level of the actor script, standard methods are defined as per usual. In this case the only method defined is the ‘put’ method; this means that any ‘get’ messages initially received will be left in the actor’s mailbox and marked at ‘unmatched’ for the time being. Once the first put message arrives it will be immediately processed, and the value being added to the buffer will be bound to the local variable ‘o’.

On the first line of the definition of put a call is made to the receive method provided by AiR. Receive is given a block – the code between do/end - in which definitions of anonymous methods are given. Anonymous method names end with a ‘!’; and bodies of anonymous methods are given inside curly brackets - an alternate way to specify a code block. In this case the only anonymous method defined is ‘get’.

Once a receive block is executed, the actor will restrict its interface to only those methods defined inside the receive block. If no matching messages are currently in the actor’s mailbox, it will put itself to sleep. When a matching message arrives - a ‘get’ in this case, the actor will be woken and the anonymous method will be executed.

Anonymous methods may contain any regular Ruby code, with a few exceptions:

- **release** must be used to return a result to the sender of the message that corresponds to the current anonymous method invocation.

- Instance variables of the actor can no longer be accessed using ‘@<name of variable>’, but should now be accessed using ‘iv.<name of variable>’.

These limitations are due to the fact that a code block is itself a Ruby object, having its own instance variables, and having the restriction that it cannot return values - unless defined in a non-standard way.

Implementing the receive block as defined above was a non trivial task. To ensure that local variables of the top level method - ‘o’ from ‘put’ in this case - are available to the block, and to allow anonymous methods to be defined in blocks - each of which can take their own arguments, required heavy use of Ruby’s meta programming facilities. Once the inner most level of a receive block has finished processing control will jump to the next line of code after the receive block; this required the use of ‘first class continuations’ which allow a specific position in the code to be saved as a ‘continuation object’ and then resumed at a later point.

**Advanced Usage**

Listing 5.4 gives a more complex example that uses nested receive blocks and method redirects.
Listing 5.4: Advanced usage of receive blocks

class AdvancedReceive < Actor
  def first(f)
    receive do
      second!(s) {
        receive do
          third!(t) {
            puts f,s,t
            top.release "first-third"
          }
        fourth!(fth) {
          puts f,s,fth
          second.release "second-third"
        }
        method :normal { puts "cleanup" }
      end
      puts "Exiting second"
      release "second"
    }
    puts "Exiting first"
    release "first"
  end
  def normal
    puts "Inside normal method"
  end
end
In this example it is clear that for anything interesting to happen first a ‘first’ message must arrive, and next a ‘second’ message must arrive. At this point a nested receive block is encountered. The nested receive block defines two anonymous methods, ‘third’ and ‘fourth’, and additionally it defines a ‘method redirect’; a redirect allows a standard method to be invoked from within a receive block, and after it has finished processing control will return to the attached code block in the receive statement - in this case the code block that prints out ‘cleanup’.

Which method will be executed inside the nested receive block is now dependent on whether a ‘third’, ‘fourth’, or ‘normal’ message arrives first.

If a ‘third’ message arrives, a short printout will be given of all the parameters received thus far, and then the code `top.release “first-third”` is called. `top.release` allows an anonymous method to return a result to the actor that invoked the top level method to which the receive block belongs - ‘first’ in this case. This result will overwrite any later release calls made for the ‘first’ method.

If a ‘fourth’ message arrives the code `second.release “second-fourth”` is executed. Similar to the top case, `second.release` allows a result to be returned to the actor that sent the message which resulted in the execution of the anonymous method ‘second’.

Lastly, if a ‘normal’ message arrives the ‘normal’ method will be executed. This will result in the text “Inside normal method” being printed, after which the code block that prints “cleanup” will be executed.

Once the inner most anonymous method or redirect has been handled, control will then jump to line 20. When this happens a printout is given, and then a result is returned to the actor that sent the ‘second’ message. A similar process now occurs for the top level receive block, and a result is given to the actor that send the ‘first’ message. In both cases, if a result has already been issued then the new call to `release` will be ignored.

5.7.2 Chords

An alternative approach, employed by languages such as Co[3] and Polyphonic C#[11], is to make use of ‘chords’ [25]. A chord is a sequence of method signatures associated with a single method body. Once all messages corresponding to the method signatures in the sequence have arrived, the method body will be executed. In addition to the use of receive blocks, AiR also allows chords to be defined. A combination of the two approaches is also possible by putting a receive block in the body of a chord.

Introductory example

To make the explanation clearer, the following code demonstrates how the unbounded buffer example could be specified using chords:
The actor waits until both a get and a put message have arrived, after which the value that was given with the put message is returned to the actor that sent the get message. The convention when using chords is that the actor which sent the message corresponding to the first method signature in the sequence is the one that gets any return value that may be sent.

This is arguably more elegant than the use of receive blocks to achieve the same feat! It does however have the limitation that no intermediate processing may occur between the receipt of messages in the sequence, as would be permitted if using receive blocks. Another limitation is that a result may only be returned for one of the messages in the sequence; this is in accordance with the original semantics of chords.

**Multi-chorded actors**

An advantage of chords over receive blocks is that once the first message in the sequence has arrived the receiving actor does not restrict its interface waiting for the remaining messages in the chord. Indeed it is possible for multiple chords to contain some of the same method signatures, and it is also possible to process messages corresponding to standalone methods while waiting for the complete sequence of messages required for a chord to be executed. To elucidate this, take the example of an actor with the following definition:

```ruby
class ChordExample < Actor
  chord.first.second { puts "inside first chord" }
  chord.first.third { puts "inside second chord" }
  def regular
    puts "Inside regular method"
  end
end
```

Now assume that messages corresponding to the following sequence of methods arrive:

first -> regular -> regular -> third

When the ‘first’ message arrives it will go unmatched and remain in the mailbox. Critically, it will not restrict any other messages from being processed.

Two ‘regular’ messages now arrive. The regular method defined upon the actor will now be invoked twice.
Lastly, a ‘third’ message arrives. Combined with the ‘first’ message already in the mailbox, this now means that the second chord is enabled and as such the corresponding method body is executed. Had a ‘second’ message arrived instead, the body of the first chord would have been executed.

To produce the same sequence of method invocations using receive blocks would have required a loop, and a potentially large number of redirects depending on the number of standard top-level methods offered by the actor - just ‘regular’ in this case.

5.8 Naming

The ability for actors to associate a system wide name with themselves and the rationale for allowing this was discussed in 4.5. This section will detail how a developer can make use of naming facilities available in AiR.

5.8.1 Specifying a nameserver

To enable actors in an AiR application to use naming operations it is first required that a nameserver be associated with the theatre that hosts said actors. A nameserver’s location consists of a hostname/port combination, and can be given as an argument to the theatre script as follows:

```ruby
Air.start :nameserver => 'localhost:1234' do
  .. theatre script body here ..
end
```

In this case the nameserver is located on the same machine as the current theatre and accepts connections on port 1234. A simple nameserver implementation is included with the AiR distribution.

5.8.2 Standard naming operations

Once a relationship with a nameserver has been defined, a unique system-wide name can be specified for an actor by calling the `register` method on a proxy for said actor, or by calling `register` from within the actor itself.

For example, given an actor proxy ‘chat_server’, we could associate the name ‘primary_chat_server’ with the actor by making the following call:

```ruby
chat_server.register('primary_chat_server')
```
It will now be possible for other actors in the system to retrieve a remote reference (proxy) for the chat server actor by making a call to the static `lookup` method on the base Actor class. For clarity the lookup should be made on the specific Actor class in question - assuming the definition is available locally. Using the aforementioned example, the code to lookup the actor would be:

```ruby
chat_server = ChatServer.lookup('primary_chat_server')
```

Where ‘ChatServer’ is the actor script class for a chat server, and ‘chat_server’ is the returned proxy that can be used for communication with the chat server actor.

In addition to being able to name actor instances, it is also possible to associate a name with a particular theatre. This is needed for migration of actors (see section 5.9). To specify a name for a theatre an additional `:name` argument is given to the theatre script, as in this example:

```ruby
Air.start :nameserver => 'localhost:1234', :name => 'theatre_one' do
  .. theatre script body ..
end
```

The concrete address (hostname/port) of a theatre can now be retrieved using a static call to `lookup` on the `Theatre` class, as shown below:

```ruby
theatre_address = Theatre.lookup('theatre_one')
```

### 5.8.3 Capability Tags

Instead of trying to lookup a particular actor instance, an actor may wish to initiate communication with any actor that can perform some specific functionality. In addition to being able to assign names to individual actors, it is also possible to specify groups of actors that perform related functions. To do this an actor registers a `tag`, which denotes a unique ability, in a particular `space`, which denotes a category. This concept was inspired by ActorSpaces [18].

Take for example an actor application that implements a search crawler (spider). Once a page has been downloaded it will need to be parsed for additional links. If there are multiple actors that can do the parsing, ignoring load balancing, it does not matter which specific one does the job. For this we could create a space called ‘search’, and the ability to parse a page for links could be represented by the tag ‘link_extraction’.

If an actor wishes to advertise the fact that it can parse downloaded pages for links it should include the following code in its constructor:
local refers to the fact that the space is managed by the theatre itself, meaning a nameserver isn't involved. When categories span multiple theatres, each sharing the same nameserver, global should be used instead.

Assuming all actors that can perform the link extraction belong to the class 'HtmlParser', the actor that has downloaded the page could use the following code to get a reference to an actor to do the parsing:

```ruby
HtmlParser.local_space_find(:one, :search, :link_extraction)
```

The :one argument specifies that only one reference is required. If proxies for all instances are required, such as when a broadcast is to be made, :all can be used to return an array of references to all actors that meet the given requirements.

## 5.9 Migration

Actors in AiR have the possibility of being allowed to move between theatres. A number of reasons why this behaviour may be desirable were given in section 3.4, and the implementation details of migration were discussed in section 4.4.

The actual code to perform such migrations is extremely simple. Should a developer wish to allow instances of a particular actor class be able to migrate between theatres, the statement `migratable :true` must be added to the relevant class. Additionally, a `migration_complete` method must be defined in said class; this will be executed when the actor has successfully migrated to a new theatre.

For example, to make a 'ProbeActor' migratable, the following class skeleton could be used:

```ruby
class ProbeActor < Actor
  migratable :true
  def migration_complete
    # re-establish contracts here
  end
  .. rest of actor definition ..
end
```

At runtime, given a proxy 'm_actor' to a migration enabled actor and a destination theatre address 'dst_theatre', the actor could be migrated to the destination theatre using the following code:

```ruby
m_actor.migrate('dst_theatre')
```
After migration the ‘m_actor’ proxy may still be used to communicate with the actor as per normal.

5.10 Exception Handling

5.10.1 Tradition approach

In a regular Ruby application an error results in the throwing of an exception, which can then be caught in a begin/rescue clause as follows:

```ruby
begin
  # code which may throw an error
rescue ExceptionType => e # store the exception as 'e'
  puts "Exception throw: "e.to_s"
end
```

This is still true in an AiR application, but with added possibility for failure. For example an exception might be raised because an actor cannot be contacted due to a communication failure, or because a request has taken too long to be processed.

5.10.2 Linked Actors

AiR additionally supports Erlang’s linked process error handling style. Using this approach actors create bi-directional links (relationships) with other actors, and in the event that an actor terminates, such as because it didn’t catch an exception, all actors linked to the terminated actor will be sent a special error message (‘exit signal’). Actors can explicitly terminate themselves through calling their `kill!` method, giving a reason for termination as an argument. A theatre may be terminated by calling its `shutdown` method, giving as an argument the time when this should happen; the shutdown call will immediately result in an exit signal being sent to all actors resident at the theatre, giving them a chance to gracefully terminate, or possibly migrate elsewhere.

To catch an exception from a linked actor an actor must define an `exit_signal` method which takes two arguments: the first gives the location of the actor that terminated, and the second gives an error message explaining what happened. An explanation of ‘normal’ says that the linked actor terminated gracefully, such as might happen if it had completed a task assigned to it. For all other explanations, if a linked actor does not catch the error message by defining an `exit_signal` method, it too will be terminated, and in turn its remaining linked actors will be informed through new error messages sent out.
This type of linked error handling is especially useful when actors naturally form master/slave relationships with each other; in such a system the master distributes work to slaves and then combines the results passed back. Should the master actor terminate, it would only be natural that the slaves should also terminate. If however a slave terminates, then the master would most likely just spawn a new slave, or possibly choose to take a different strategy, and so should make sure to catch any exceptions thrown.

Listing 5.5 shows a very basic actor application in which one actor, the master, is tasked with calculating the squares of all numbers up to some value \( n \). To perform this task the master actor spawns \( n \) workers, each of which calculates the square of a single number. If an exception is thrown by one of the slaves it will result in a new slave being created in its place.

Such an example is merely for purposes of demonstration and would not be wise to implement in practice, as the overhead of spawning a new actor and sending messages between the master and slaves would be far greater than the time required to calculate each square. \texttt{get_address} is a method that can be called on a proxy object to get the concrete address (theatre address/unique id) of the actor. \texttt{find_index} is a standard Ruby method defined upon arrays which returns the first index of the array in which the condition specified in the block is true - in this case the array index of the slave that died.

### 5.11 Summary

The AiR library provides Ruby developers with a domain specific language for writing concurrent applications. This chapter has explored the various language features and constructs supported.

A class based approach has been taken for the definition of actor scripts; the behaviour of an actor is defined via standard Ruby method, and actors may posses local state - i.e. they are ‘serialized’ actors. When interface restriction is required, such as in the unbounded buffer example that was described, it is possible to define chords and receive blocks to allow for this.

Actors communicate with each other through the use of proxy objects. Method invocations on such proxies transparently result in the generation of an appropriate message, which is then dispatched to the destination actor that the proxy masks. A number of alternative types of messages may be specified, and a fluent interface is provided to make it as easy as possible to specify the particular messaging semantics required. Futures are one of the means by which values can be returned once a message has been processed; a two-way message send is non-blocking, with a placeholder - the future - being returned in place of the actual result. When the time comes for the actual result to be used, a process of synchronization take places, possibly resulting in the actor that requires the value being put to sleep if it is not yet available.
Listing 5.5: Linked Actor Exception Handling

```ruby
class Master < Actor
  def init(n)
    @slaves = []; @results = [];
    n.times do |i|
      @slaves[i] = Slave.spawn(i, actor_self)
      slave.link(actor_self)
    end
  end

  def submit_result(i, square)
    @results[i] = square
    # check if all results returned, do extra processing here...
  end

  def exit_signal(source, reason)
    unless reason == "normal"
      i = @slaves.find_index do |slave|
        slave.get_address == source.get_address
      end
      @slaves[i] = Slave.spawn(i)
    end
    @results[i] = Slave.spawn(i)
  end
end

class Slave < Actor
  def init(i, master)
    master.submit_result(i, i*i)
    kill!("normal")
  end
end

Air.start do
  master = Master.spawn(999)
  # get the result and do additional processing..
end
```

A number of naming operations can be made use of by actors, assuming a nameserver has been specified by the theatre in which they are resident. An actor can register a unique system-wide name, which can subsequently be used by other actors to lookup the registered actor’s mailbox address. The ability for actors to tag themselves, a unique feature inspired by actor spaces, is also available. Tagging allows an actor to advertise its capabilities to the system, and a lookup operation that works with tags rather than names is also offered; using this, it is possible to retrieve references to one or more actors that have registered a particular tag, in a particular space (category).

Language level support for migration is provided. Given a destination theatre address an actor can make a single call and have itself relocated. A special method can be defined upon such actors which will be automatically invoked upon arrival at their new home, providing a means to re-establish relationships broken by the move.

Lastly, the linked actor concept of Erlang is used to provide support for distributed exception handling. This is especially useful in systems where actors form master/slave relationships.

Appendix A.3 gives a full API listing and grammar for the language that has been discussed here.
Chapter 6

Evaluation

When developing software it is not enough to simply implement the system and have done with it. Instead, the value of the implementation must first be gauged through an appropriate evaluation.

There are numerous criteria against which a project’s success can be measured, in particular the evaluation for this project covered the following criteria: correctness, performance and effectiveness.

6.1 Correctness

As new functionality was added to the AiR package it was important to ensure that such functionality behaved as expected, and, more importantly still, to check that existing functionality was not adversely affected. To achieve such correctness testing two approaches were used; first, unit tests were defined to check the outputs generated by the library against pre-defined data sets; and second, a number of medium to large scale sample applications were developed to ensure that the different components implemented worked in a cohesive manner.

Ruby provides the ‘Test::Unit’ module for implementing unit tests, and as such this is what was used to unit test AiR. Listing 6.1 gives a flavour of what unit tests written with Test::Unit in the context of AiR look like. In this unit test case, the FluentMethod class is being tested. This is a helper class used by Actor proxies, and its task is to generate messages based upon a sequence of chained method invocations. Section 5.4 details the different messages that can be created.

The first thing to note in this example is that a class called ‘MockProxy’ has been created. This is used to create mock objects[46] representing actor proxies. The reason why a real proxy object was not used is because the aim of unit testing is to test the application at the micro level, isolating individual
Listing 6.1: Example AiR Unit Test Class

```ruby
require "test/unit"
require "./:lib/language/fluent_method.rb"

class MockProxy
  attr_reader :message
  def send_actor_message(message)
    @message = message
  end
end

class TestFluent < Test::Unit::TestCase
  def setup
    @mock_proxy = MockProxy.new
    @fluent = FluentMethod.new(@mock_proxy)
  end
  def test_one_way
    @fluent.>.goodbye
    message = @mock_proxy.message
    assert(message[:one_way]) # is it True?
    assert_equal(message[:method], :goodbye)
  end
  def test_two_way
    @fluent.query_price
    message = @mock_proxy.message
    assert(!message[:one_way]) # is it False?
    assert_equal(message[:method], :query_price)
  end
  def test_priority
    @fluent.*.check_status
    message = @mock_proxy.message
    assert(message[:priority])
    assert_equal(message[:method], :check_status)
  end
  def test_composed
    @fluent.>.*.relocate
    message = @mock_proxy.message
    assert(message[:priority])
    assert(message[:one_way])
    assert_equal(message[:method], :relocate)
  end
end
```
components from each other. Internally, FluentMethod objects make a call
to the 'send_actor_message' method defined upon an actor proxy - given at
initialization, and therefore this seemed an appropriate place to intercept the
generated messages. attr_reader is a Ruby class method that automatically
creates a getter method for the given instance variables, in this case creating a
getter method to access the intercepted message.

The actual unit tests are defined in the class TestFluent, which extends the
built-in TestCase class. ‘setup’ is a special method that is called before each
unit test is executed; its job is to create any common objects needed for the
tests, in this case an instance of FluentMethod and an accompanying proxy
mock object. Looking at the ‘test_one_way unit test’, we observe that on the
first line a method sequence is invoked upon the fluent object representing the
fact that a one-way message asking for a method ‘goodbye’ to be invoked should
be generated. The message output by the fluent object is intercepted by the
mock, and the message is then read into the local ‘message’ variable. At this
point the assertion methods provided by Test::Unit, a full listing of which can
be found in [51], are made use of to check that the contents of the message do
indeed encode for a one-way message asking for ‘goodbye’ to be invoked.

It was also mentioned that a number of sample applications were created to
test the system at the macro level. The complete set of these can be found in
the ‘/demo’ folder of the AiR distribution, and some notable examples, each of
which is discussed below, are:

- An implementation of the Monte Carlo approximation of π
- A distributed chat application
- A news website (including web server)

6.2 Performance

6.2.1 Execution Speed

The actor model inherently exploits parallelism, and therefore it is only natural
that programs developed using an actor library should have performance gains
over sequential code attempting to achieve the same purpose. It is therefore
important to set-up a number of experiments to test whether such gains are
observed when using the library.

One problem that lends itself naturally to parallelism is an approximation of π
that uses the Monte Carlo method. The Monte Carlo method refers to a group
of algorithms that work through the repeated generation of random numbers
and subsequent sampling of said numbers, observing the fraction that conform
to some pre-defined properties [54].
Using this approach to calculating $\pi$, we must first imagine a circle of radius one enclosed in a square with edge length two. The area of a circle is $2\pi r^2$, and in this case the area of the enclosing square is $(2r)^2$; this means that the ratio of the circle’s area to the square’s area is $\pi/4$, and re-arranging we find that $\pi = 4 \times \text{area of circle}/\text{area of square}$. The ratio can be calculated by generating random values for $x$ and $y$ between zero and one and calculating the fraction of points which fall within one quadrant of the circle, with the origin being at the center of the circle. Such calculations can be performed in parallel, as the generation and checking of a point does not rely on any data other than the radius of the circle which is fixed. Figure 6.1 illustrates the problem, where an ‘x’ symbol represents a random point.

Two implementation of this algorithm were created. The first is a standard, single threaded, Ruby application. The second is an actor-based solution that uses AiR; in this solution a master actor spawns two worker actors, each of which runs the algorithm for half the regular number of points, after which the results are returned to, and combined by, the master actor to give the final approximation. The code for each of these implementations, including the logging of results, can be found in Appendix A.4.

The implementations were then benchmarked against each other for increasing numbers of randomly generated points. The machine that was used to perform the benchmarks ran the Linux operating system, had 2GB RAM, and used an Intel(R) Core(TM)2 CPU 6420 @ 2.13GHz processor.

Figure 6.2 displays the results of these benchmarks, and a tabular view of the same data can be found in Appendix A.4.3.

The chart shows that the AiR implementation performs significantly better than the regular Ruby implementation for large numbers of point (iterations). The speed-up was not quite 200% - the optimum speed-up that could be expected on a dual core system, however it was not far off; the explanation for not achieving
the optimum case is most likely due to the fact that the actor system does not have exclusive use of the processor, and because the generation and passing of messages incurs a slight cost.

For very small numbers of points the single threaded Ruby version actually performed better than the parallel version. This suggests that AiR is not well suited for extremely fine grained problems, as for such cases the cost of message passing outweighs the cost of the actual computation being performed. This is a general problem with any message passing system, and is not restricted to AiR exclusively. One possible way to mitigate this problem would be to ensure that actors that pass high volumes of messages to each other, each of which only results in a very small amount of computation, should be located on the same slave theatre; this would mean that TCP need not be used to pass the messages, but instead they could be passed locally - this does not however remove the cost incurred by message generation.

The choice to not benchmark the execution speed of AiR against that of other actor libraries was intentional. The reason for this is that Ruby’s main strengths are the rapid development times it affords and its powerful dynamic programming capabilities, but not its execution speed – being far slower than languages such as C++. It was therefore decided that the most appropriate comparison would be with Ruby itself, the language upon which AiR is built.
With the current exponential increase in cores that modern processors are seeing it is increasingly important to break problems down into smaller chunks - it is also beneficial on single cores to avoid problems like blocking I/O. The scalability of the library, in terms of how many actors can simultaneously exist on a single machine, was therefore tested.

In [21], Joe Armstrong, creator of Erlang, sets out a challenge which is defined as follows:

1. Put N processes in a ring.
2. Send a simple message round the ring M time.
3. Increase N until the system crashes.
4. How long did it take to start the ring?
5. How long did it take to send a message?
6. When did it crash?

It was thought that an implementation of such a system would act as an excellent way to measure the scalability of the library, and at the same time would provide an evaluation of message-passing speed afforded by AiR. The AiR implementation of the challenge is given in Appendix A.5.1, and the benchmarking of the implementation was performed on the same machine as was used for the Monte Carlo approximation of $\pi$.

It should be noted that the number of slave theatres used was increased to sixteen, from the regular two for dual core systems, as this appeared to improve performance in extreme cases; this can be attributed to threads within a particular virtual machine instance not getting a large enough timeslice to fully execute when only a small number of slaves were used. Each slave theatre uses around 10MB of memory initially, and as such, using sixteen was not at all unrealistic for a system with 2GB RAM.

Figure 6.3 details the total time it took to start the ring for an increasing number of actors, with a tabular view given in Appendix A.5.2. In most realistic systems it is unlikely that more than a few hundred actors would ever be started at any one time, and often it is the case that only a handful need to be started. From this perspective the results are very impressive, with it taking a mere 1.7 seconds to spawn 1000 actors. These results help to justify the decision that was made to use green threads (4.3.1) in the implementation of AiR.

The message passing times are shown graphically in Figure 6.4, and a tabular view is given in Appendix A.5.3. Additionally, the message-passing times for STAGE, as outlined in [24] are displayed. Python, the language upon which
STAGE was built, is not too dissimilar from Ruby - although it is known for being somewhat faster, and therefore this was seen as being a good language to make a comparison against.

It should be noted that the methodology used in the STAGE evaluation was slightly different, with the average message passing time including the time taken to spawn actors and to then pass a token around the ring 100,000 times. In AiR a token is passed around the ring once and the average message passing times do not include the time taken to initially spawn the actors. Out of curiosity, an alternate version was created in AiR that followed the STAGE methodology, the full benchmarks for which are not given, and in this version it took an average of 0.0005362 second to pass a message in a ring of size 5000.

The results of this test show that even as the number of actors in the ring increased substantially the average message passing time was not adversely affected. It was only after having 20,000+ actors in the ring that a substantial decrease in performance became evident, with a ring of 30,000 actors being unable to pass the token within a five minute period - after which point the test was terminated.

From the figure, there is a clear jump in message passing time for the case of 1000 messages, after which point the average time falls again. The test for this case was repeated a number of times, and the results were near identical in each. This would suggest that the abnormality lies with the Ruby thread scheduler. Although undesirable, the sudden spike is far from detrimental to performance, and results are much more consistent for larger number of messages.
6.3 Effectiveness

Software developed in Ruby is generally developed far quicker than in other languages, and with shorter, more elegant code. It is crucial that such strengths are not lost when using AiR to develop concurrent applications. Additionally, it is also important to ensure that real-world applications can be developed with the library, and not just toy examples. Two applications were developed to try and get a better understanding of how well AiR holds up in these regards.

The first was a distributed chat application, which was additionally developed in Java. The aim of this experiment was to compare and contrast the relative complexities of the two implementations. Such a comparison is no trivial task as there are no clear guidelines on how to do this, there are however some common rules of thumb that can be invoked, such as to count the number of lines of code for each implementation.

The second, more involved, application developed was that of a news website - complete with the HTTP server required for interaction with browser users. As such applications are abundant in the Ruby community, failure to be able to develop this medium-scale example in AiR would highlight serious shortcomings with the library for use in real-world scenarios.
### 6.3.1 Distributed Chat Application

To develop a chat application requires the logic for two classes of components to be implemented: a server into which clients can connect, and the clients themselves. The job of the server is to maintain a registry of the clients participating in the chat session, and to broadcast any message received from one of the clients to the entire group. Client applications must be able to handle user input, such as from a keyboard, and simultaneously must be capable of displaying any messages received from the server.

The Air implementation of the chat application is given in Appendix A.6.1. The server and clients are each represented by an actor, with an additional built-in actor called ‘KeyboardActor’ being used to take user input. Should keyboard input be received by this actor it is forwarded to the ‘user_input’ method of the corresponding chat client.

The Java implementation is given in Appendix A.6.2, and was based upon code from an online tutorial. In this case the client is encapsulated in the class ChatClient; with user input being handled by the static main method, and input from the server being handled by a separate thread. The server logic is split between the ChatServer and clientThread classes; the former is responsible for accepting connections from clients and broadcasting messages, whilst instances of the latter are used to take input from a particular client.

In terms of lines of code occupied by each implementation, the Air version comprises of 53 lines, whereas the Java version comes in at 104 lines. Although not always a good indication of relative complexity, the difference in this instance is quite substantial given that both provide identical functionality, and therefore this has to count in Air’s favour.

To initiate communication between the client and server in the Air version requires a single line of code for the lookup of the server by the client, and subsequent communication happens transparently thanks to the use of active objects, appearing no different from regular method invocations. To preserve the non-blocking semantics offered in the Air implementation, the Java version resorts to low-level socket operations, with numerous input and output stream objects needing to be created before even the simplest items of data can be exchanged. This mixing of application and communication logic, and the general verbosity of Java, makes it harder to understand the core intent of the implementation. Additionally, in the Java version the chat server is contacted using a static address, which would mean that should the location of the chat server be altered a change in the client side code would also be needed; in the Air version the naming operations are not dependent on the physical location of an actor, and thus it is much more flexible in this regard.

The last major difference between the two is that the Java version makes use of threads at both the server and clients. These are needed to allow for multiple clients to be connected to the server at one time, and for simultaneous input
and output at the client. In the AiR version, no extra code was needed at the server to support for multiple clients, and at the client only one line was needed to allow for simultaneous input and output - the creation of a keyboard actor.

### 6.3.2 News website

One area that Ruby is particularly well known in is that of web development. This is due in large part to the Ruby on Rails web framework, which in turn owes much of its success to Ruby’s rich meta-programming facilities. The web is also of particular interest to the field of concurrency as at any one time it is likely that a popular website is being accessed by multiple users, and as such incoming requests should ideally be handled in parallel to provide maximum responsiveness. It was therefore decided that it would be appropriate to implement a web application in AiR, specifically a news website. This section gives an overview of the applications development process and highlights any areas where problems were encountered.

The runtime architecture of the application that implements such a site is shown in figure 6.5. The most important component in the application is the HTTP server actor; this provides a way for browser users to access the site from the outside world. Once a request has come in from a browser user it is assigned to an appropriate handler actor, depending on the type of request made; for example, a request might ask that an XML representation of a news article be returned, in which case an XML handler would be chosen. The handlers retrieve the requested article(s) from a back-end database, after which point the data is processed and returned in the desired format using an HTTP response message.

Implementation of the HTTP server actor relied heavily upon Ruby’s socket libraries. At the core of the server actor is a TCPServer object that listens for connections on port 80, generating socket objects for sending responses to clients that decide to connect in. The use of Ruby’s socket library from within an AiR application did not cause any conflicts with AiR’s own communication.
layer; this is important as most real-world actor applications are likely to need to interface with non-actor entities at some point, most often through the use of sockets.

After a connection has been established with a client the HTTP request message is read from the resulting socket object, and matched against a series of regular expressions; these are used to determine the data being requested, and the format in which the data should be served in the response message. Once the format of the request has been determined this is used in combination with AiR’s tagging (5.8.3) facility to select an appropriate handler for the job. The parallelism in this application comes from the fact that there are multiple handlers in the system, possibly with overlapping functionality - for dealing with similar requests that come in at the same time. Additional handlers can join the system at any time, needing only to register an appropriate tag with the system, after which they will automatically be made available to the server when it searches for a handler to carry out a request. Due to the distributed nature of AiR it is not a requirement that handlers reside on a single machine, and so theoretically a large network of computers could collaborate in processing the requests with minimal effort on the developer’s part.

The initial intention when implementing the application was that once a handler had been selected it would be given the request message along with the socket to send the response over, and after this point no additional interaction would be necessary until further requests arrived. This way of thinking was rather naive - and no doubt a remnant of working with threaded applications - as it is not possible to marshaled a socket, one of the requirements for passing objects between actors. The explanation for this is obvious: a socket represents an endpoint of a connection between two distinct applications, and therefore it is only natural that it cannot be passed between actors, each of which need not reside in the same virtual machine instance, or indeed even on the same host.

To solve the socket problem it was decided that the HTTP server would act as a sort of hub for sockets, creating them and providing an interface to send data over a specific socket. Using this approach, when requests are dispatched to a handler, the handler is also given an ID representing the corresponding socket, and when finished a callback is made to send the response message. The callback is transparent to the handler, thanks to the use of message chains (5.6).

An alternate approach would have been to create a ‘socket actor’ and pass this to the handler; the underlying implementation would have been very similar though, with the socket actor encapsulating the ID and performing the callback to the HTTP server actor.

Three handler classes were created, for the following data formats: RSS, XML, and HTML. To generate the responses the handlers need to first retrieve the actual articles from a database actor. They do this by using AiR’s naming facilities to retrieve a proxy for communication with the database, after which a number of simple method invocations can be used to retrieve the required data. The generation of HTML responses was very straightforward, requiring
a simple iteration through the data provided by the database and addition of a
minimal number of HTML tags. For generation of XML and RSS responses the
process is a little more involved and it was decided that two Ruby libraries would
be used for these tasks: REXML[12] and RubyRSS[14]. The incorporation
of these libraries into an AiR application was no harder than it would be to
incorporate them in a regular Ruby application; simply requiring an import,
after which actors can make calls directly into the library methods. Again
it should be noted that this is important as any real-world AiR application
is likely to at some point require the use of third party Ruby libraries; had
this interaction not been possible it would have severely limited the scope of
applications for which AiR could be realistically used.

To provide a level of fault-tolerance, handlers can be given an optional backup
theatre’s address as an initialization parameter. If it happens that a call is made
to shutdown the theatre on which a handler is resident this will be intercepted
by its exit_signal method, resulting in the handler migrating to the backup
teatre if one was given.

The database actor provides a container for articles and offers a small number
of operations that can be performed, namely to: add articles, remove articles,
and retrieve articles. The actual implementation of this component is largely
uninteresting, so shall not be discussed further. As the database could act as
a single point of failure, and as a bottleneck for the handlers, for a large scale
deployment it might be worth making multiple replicas of the database. Using
AiR’s tagging abilities this could be done in a manner similar to that used for
handlers, and with updates to all replicas being made through a broadcast. In
most scenarios, however, the computation performed at the handlers would far
outweight the cost of retrieving the data, and therefore having a single database
would be unlikely to be prohibitive other than for very large scale websites.

An implementation of this application in pure Ruby would no doubt have com-
prised of a similar set of components, possessing similar internal logic. A thread
would be needed for each executing handler, however this usage would be fairly
trivial. Assuming that the server and all handlers existed in the same virtual
machine instance, this would actually have resulted in a simpler implementation
from the perspective of passing socket objects around. If however the applica-
tion was to scale to multiple cores and offer the runtime extensibility of the
AiR version this would have required a great deal of additional work, and the
problem of passing sockets would once again have to be dealt with - this time
with additional complexity.

6.4 Conclusions

This chapter has seen AiR being evaluated from a number of different view-
points. The quantitative evaluations have shown that AiR offers serious per-
formance gains over traditional single process Ruby applications when multiple
cores are present - given the right granularity of the solution. Additionally, AiR’s ability to scale to large numbers of actors whilst at the same time maintaining operational integrity was demonstrated.

From the qualitative perspective, a number of examples were implemented and discussed. These showed that AiR offers a number of clear benefits over more traditional languages that are used for writing concurrent applications, such as Java. The ultimate way to truly test the effectiveness of a language though is to actually let outside developers try it out; one hope of this project is that the codebase could be potentially turned into an open-source project, and therefore if this does take place I look forward to seeing how AiR is received by the Ruby community.
Chapter 7

Conclusion

This report started by giving an investigation into the actor model, a model in which concurrently executing entities known as actors that each have their own private state and communicate via asynchronous message passing are the main unit of decomposition. The main body of the report discussed the development of AiR, an actor-based library for Ruby that provides a domain specific language for writing concurrent applications, and a supporting runtime environment for such applications. Lastly, an evaluation of AiR was given in which the library was looked at from a number of different perspectives.

Through the development of numerous example applications using AiR, and other actor languages such as Erlang, the actor model has proven itself to be a far simpler model to work with than that of threads and shared memory. AiR itself is built using threads - something hidden from developers using the language, and many of the common problems such as deadlock reared themselves during its development. The pure actor model (2.1), in which ‘everything is an actor’, was seen to be too low-level at times, and as such some concessions occasionally had to be made; for example, actors can use standard Ruby data types, there is no need to use an ‘Integer’ or ‘String’ actor - although these could be implemented if really desired.

Coming into this project I had a surface level understanding of Ruby, thanks to two internships that involved using Rails, and a side project worked on with a few friends one summer holiday. It was only after starting development on AiR that I began to fully appreciate the power of Ruby in regards to its meta-programming facilities; these proved indispensable, and made tasks such as method call interception trivial. From the perspective of developing an embedded language, Ruby has proven itself to be an ideal language.

What was not fully understood initially was the relative slowness of Ruby in relation to other object-oriented languages such as Java and C++. The Great Win32 Computer Language Shootout [7], gives a good insight into this by implementing a number of common problems in Ruby and other languages and
then running benchmarks on each. Fortunately this is a problem that is being actively tackled by the Ruby community. The latest ‘experimental version’ of Ruby, 1.9, has serious performance gains over previous versions, and a number of groups, such as the Rubinius team [13], are working on alternative implementations of the Ruby virtual machine which boast similar speed-ups.

Taking performance into account, when developing truly large scale applications that are highly performance critical a language such as Erlang would probably be a better choice than AiR; this doesn’t mean that AiR is useless in such scenarios however, and it is the authors view that AiR could prove to be extremely useful for rapid prototyping of such a system, before making a transition to a more performance oriented language. It is hoped that the examples detailed in the evaluation section showed that AiR met the original aim of retaining the ease-of-use and elegance of its mother language Ruby. The evaluation also showed that AiR applications can potentially run much faster than regular Ruby applications, thanks to the multiple virtual machine approach used.

In chapter 3 a number of requirements of the language were outlined. With the exception of security, I feel that AiR generally performs strongly in relation to these. The use of active-objects proved to be an excellent way to fuse the actor model with the object-oriented programming paradigm, and the novel approach of using chords for interface restriction proved an elegant substitute for explicit receive blocks - something also implemented to provide flexibility and because of their differing semantics.

Distribution in AiR is inherently supported and is largely transparent to the developer, meaning that dynamically evolving distributed systems can easily be created by making simple use of the naming facilities provided. The use of tagging is particularly notable as it is something I have not seen before in another actor implementation, and it demonstrated its worth when implementing the web application of section 6.3.2.

AiR provides an implementation of weak mobility, allowing actors to move between theatres. It was originally intended that strong mobility be implemented, however limitations inherent to Ruby made this infeasible. On the topic of mobility, I would have liked to have provided an implementation of load balancing with the AiR library, however it was decided that to do this properly would be a project in itself.

The reason for not dealing with the issue of security was similar to that of load balancing: protecting an application built in a dynamic language such as Ruby from malicious users, especially in a distributed environment with support for mobility, is no trivial task. The strong support for meta-programming in Ruby is a bane from the perspective of security as it means that the experienced developer can almost always find some way to hack around restrictions that have been implemented. For these reasons it was decided that since the central focus of the project was not on providing a secure implementation, it was an issue that should best be left to a separate project.
7.1 Further Work

As the development of any large application progresses, it will usually be the case that a number of possible extensions are highlighted. This section discusses some areas for which further work could potentially be of benefit to AiR.

It was previously mentioned that security and load balancing were issues that were not tackled in the current implementation. The reason for this was lack of time, not because they are unimportant issues, and therefore a further investigation into these two areas would be of great utility. Security is naturally going to be important for any application which interfaces with the outside world, as exposure of an unsecure system could put an entire companies network at risk of attacks by malicious users. Load balancing on the other hand is important as it helps to makes sure that computational resources are not over or under used.

When developing applications using AiR, although it was not a major roadblock, due to the language’s distributed nature it sometimes took longer to pinpoint the source of errors than would be the case for a standalone Ruby application. The development of an actor-based debugger could greatly help in such cases. A related extension would be the development of a visualization component that would provide details of the actors in a running system and the activities being performed at each.

Ruby 1.9, the latest ‘experimental version’ for developers provides a new approach to the issue of concurrency, namely ‘fibers’. An investigation into the similarities and differences between fibers and green threads could result in a possible alternative way of giving each actor a unique thread of control. From my limited understanding of fibers, unlike green threads they are co-operatively scheduled, meaning that without intervention a task would run to completion before another fiber is permitted to executed; clearly this is undesirable and should be taken into account when performing such an investigation.

An alternative, shared memory based approach, to the actor model is software transactional memory (STM) [49]. The major benefit of this approach over the actor model is that when performing a group of operations that require the exclusive use of shared resources, such as is needed when transferring money from one bank account to another, it is possible to simply mark these operations as ‘atomic’ and the underlying locking is performed transparently. In the actor model however, it would be required that messages first be sent to these shared resources to ensure exclusive access before proceeding. Therefore, an investigation into how transactions can best be dealt with in the context of an actor system would certainly be of value. In [30] the ‘Transactor’ project is described, which from initial inspection appears to be highly relevant to this topic area and so should be explored further. It would also be worth looking at Mnesia [9], Erlang’s distributed database, to see how it has dealt with the issue of transactions.
7.2 Closing remarks

Overall I believe this project has been a success, with the majority of the original goals being fulfilled. The evaluation showed that AiR applications are capable of performance gains over their Ruby brethren, and that the language and features AiR provides can help ease the development process when writing concurrent applications. Obviously, as with any piece of software, there is always more work that can be done and new directions to be explored, and therefore I would be happy if this could happen for AiR through an open-source project or otherwise.
Bibliography


[37] Philipp Haller and Martin Odersky. Event-based programming without inversion of control.


Appendix A

Syntax

A.1 Scala

Defining an Actor

```scala
class <Actor name>(<state variables>) extends Actor {
  def act() {
    <Actor body>
  }
}
```

Actors in Scala are created by extending the provided 'Actor' class and implementing the abstract method 'act'.

Sending Messages

```scala
<target> ! <message>
```

Performs a basic asynchronous message send, sending 'message' to the Actor 'target. Scala supports the use of Futures <<see later section>> for returning values computed by the target.

Receiving Messages

```scala
receive {
  case pattern1 => ..
  case pattern2 => ..
  ...}
```
Suspends the Actor, waiting for messages that match the given patterns. If a message arrives which matches a pattern then the statements following '=>' will be executed.

A.2 SALSA

Defining an Actor

```
behavior <actor name> { <behavior body> }
```

Actors are defined through the 'behavior' keyword. The body of such a behavior definition consists of a collection of methods and instance variables.

Sending Messages

```
<targer> <- <message>
```

Sends the given message to the specified target. The message takes the same form as a method invocation, e.g. print(“hello”).

Receiving Messages

```
<return type> <method name>(<args>) { <method body> }
```

Since SALSA follows the active object pattern, receiving messages simply consists of defining an appropriate method.

A.3 AiR API & Grammar

Note: Please refer to the language chapter (5) of the report for the semantics of the methods and constructs listed below, if they are not intuitive.

Actor script definition:

```
class <actor script> < Actor
  migratable :true?
  def <init>(<arguments>)?
    <AiR/Ruby statements>*
  end
  def <method name>(<arguments>)*
```
```ruby
<AI/Ruby statements>*
end

def migration_complete?
  <AI/Ruby statements>*
end

def exit_signal(<actor proxy>, <reason>)?
  <AI/Ruby statements>*
end

chord.<method one>.<method two>..<message n>(<arguments>)* do
  <AI/Ruby statements>*
  release(<value>)*
end

Theatre script definition:

Air.start <initialization flag>* do
  <AI/Ruby statements>*
end

Initialization flags:
:nameserver => <address>
:name => <theatre name>

Actor instantiation:

<actor proxy> = <actor script>.spawn(<arguments>)

Message passing:

<future variable>? = <actor proxy>.<customization flag>*.<method name>

Customization Flags:
>'>' or 'one_way' - one way messaging
'**' or 'priority' - priority messaging

Note: default message passing style is two way.

Futures:

<future variable>.sync
<future variable>.available
<future variable>.<method name>

Future Groups:
```

85
<future group> = FutureGroup.new

<future group>.selective_wait(<cardinality>) do |<future object>|
  <AiR/Ruby statements>*
end

Cardinality: ':one' or ':all'

Message chains:

<actor proxy one>..<method name>!(<arguments>)
<actor proxy two>..<method name>!(<arguments>)
..,
<actor proxy n>..<method name>!(<arguments>).<terminator>

Terminators: 'end' or 'return'

Special argument available: 'token'

Receive blocks:

receive do
  <anonymous method>!(<arguments>) {
    <AiR/Ruby statements>*
    <anonymous method>.release(<value>)*
    release(<value>)*
  }*

  method <method name symbol> {
    <AiR/Ruby statements>*
  }*
end

Naming:

register(<actor name>)
<actor proxy>.register(<actor name>)

<actor proxy> = Actor.lookup(<actor name>)
<actor proxy> = <actor script>.lookup(<actor name>)
<theatre address> = Theatre.lookup(<theatre name>)

Note: see also theatre script section.

Tagging:

local_space_add :name => <space name>,
  :tags => [<tag>*]
global_space_add :name => <space name>,
Migration:

```ruby
<actor proxy>.migrate(<theatre address>)
```

Note: see also actor script section.

Exception Handling:

```ruby
<actor proxy>.link
kill!(<reason>)
Theatre.main.shutdown(<seconds from now>)
```

Note: see also actor script section.

### A.4 Monte Carlo π approximation

#### A.4.1 Sequential Ruby Version

Listing A.1: Monte Carlo Pi - Ruby

```ruby
def calculate(iterations)
    circle_count = 0
    start = Time.now
    1.upto(iterations) do |i|
        x = rand; y = rand;
        circle_count += 1 if Math.sqrt(x*x + y*y) <= 1
    end
    time_taken = Time.now.to_f - start.to_f
    puts "Time Taken: #{time_taken}"
    puts "Pi approximation: #{(circle_count.to_f/iterations.to_f)*4}" $result_file.puts "$result_file.puts "#{iterations}, #{time_taken}"
end
$result_file = File.new("pi-standard.txt", "w+")
[100,1000,10000,100000,1000000,4000000,7000000,10000000].each do |i|
calculate(i)
```

87
A.4.2 Parallel AiR Version

Listing A.2: Monte Carlo Pi - AiR

```
require File.dirname(__FILE__) + "/../lib/air.rb"

class PiMaster < Actor
  def init
    @worker_count = 2; @workers = []; @max=10000000;
    @turn = 0; @tries = 3; @total_time = 0;
    @result_file = File.new("#{File.dirname(__FILE__)}/pi-actors.txt", "w+")
    @worker_count.times { |i| @workers[i] = PiWorker.spawn(actor_self)}
  end

  def run_test(iterations)
    sleep 0.1
    work_per_actor = iterations/@worker_count
    @iterations = iterations
    @results = 0; @total = 0;
    @start = Time.now
    @worker_count.times { |i| @workers[i].> calculate(work_per_actor)}
  end

  def submit_result(result)
    @results += 1
    @total += result
    if @results == @worker_count
      time_taken = Time.now.to_f - @start.to_f
      @total_time += time_taken
      @turn += 1
      if @turn % @tries == 0
        puts "Time taken: #{@total_time/@tries}"
        puts "Pi approximation: #{4*(@total.to_f/@iterations.to_f)}"
        @result_file.puts "#{@iterations},#{@total_time/@tries}"
      end
    end
    unless @iterations == @max && @turn % @tries == 0
      if @turn % @tries == 0
        @total_time = 0
      if @iterations < 1000000
        actor_self.> run_test(@iterations*10)
      end
    end
  end
end
```

88
class PiWorker < Actor
  def init(master)
    @master = master
  end
  def calculate(count)
    circle_count = 0
    1.upto(count) do |i|
      x = rand
      y = rand
      circle_count += 1 if Math.sqrt(x*x + y*y) < 1
    end
    @master.>.submit_result(circle_count)
    return nil
  end
end

Air.start do
  master = PiMaster.spawn
  sleep 2 # spawning time not included in result
  master.run_test(100)
end

A.4.3 Benchmark results

<table>
<thead>
<tr>
<th>Points</th>
<th>AiR (seconds)</th>
<th>Ruby (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.00157</td>
<td>0.00048</td>
</tr>
<tr>
<td>1000</td>
<td>0.00577</td>
<td>0.00383</td>
</tr>
<tr>
<td>10000</td>
<td>0.02187</td>
<td>0.03741</td>
</tr>
<tr>
<td>100000</td>
<td>0.17522</td>
<td>0.28150</td>
</tr>
<tr>
<td>1000000</td>
<td>1.62307</td>
<td>2.80307</td>
</tr>
<tr>
<td>4000000</td>
<td>6.43503</td>
<td>11.20711</td>
</tr>
<tr>
<td>7000000</td>
<td>11.22527</td>
<td>19.64369</td>
</tr>
<tr>
<td>10000000</td>
<td>16.04913</td>
<td>28.07266</td>
</tr>
</tbody>
</table>
A.5 Armstrong Challenge

A.5.1 AiR implementation

Listing A.3: AiR implementation of Armstrong challenge

```ruby
require File.dirname(__FILE__) + "/../lib/air.rb"

class CounterActor < Actor
  def init(id, start_time)
    @id = id
    if id < 2000 # the number of actors in the ring
      @next = CounterActor.spawn(id+1, start_time)
    else
      # Time taken to spawn the actors
      puts "Finished spawning: " + ((Time.now.to_f - start_time.to_f).to_s)
    end
  end

  def start(start_time)
    @next.>>.inc(1, start_time)
  end

  def inc(num, start_time)
    if @next
      @next.>>.inc(num+1, start_time)
    else
      # Average time to send a message
      puts "Finished sending: " + ((Time.now.to_f - start_time.to_f)/@id.to_f).to_s
    end
  end
end

Air.start do
  ca = CounterActor.spawn(1, Time.now)
  # sleep an appropriate amount of time,
  # to separate the spawn calculation from message passing calculation
  ca.start(Time.now)
end
```
A.5.2 Spawn time benchmark results

<table>
<thead>
<tr>
<th>Ring size</th>
<th>AiR - Time taken (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.01863</td>
</tr>
<tr>
<td>100</td>
<td>0.01863</td>
</tr>
<tr>
<td>1000</td>
<td>1.77877</td>
</tr>
<tr>
<td>2500</td>
<td>4.43784</td>
</tr>
<tr>
<td>5000</td>
<td>10.26020</td>
</tr>
<tr>
<td>7500</td>
<td>14.82877</td>
</tr>
<tr>
<td>10000</td>
<td>21.50970</td>
</tr>
<tr>
<td>12500</td>
<td>28.57929</td>
</tr>
<tr>
<td>14285</td>
<td>32.04936</td>
</tr>
<tr>
<td>20000</td>
<td>49.17066</td>
</tr>
<tr>
<td>30000</td>
<td>198.30087</td>
</tr>
</tbody>
</table>

A.5.3 Average message passing time benchmark results

<table>
<thead>
<tr>
<th>Ring size</th>
<th>AiR - Time per message (seconds)</th>
<th>STAGE - Time per message (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.00047979</td>
<td>0.00063498</td>
</tr>
<tr>
<td>100</td>
<td>0.00034731</td>
<td>0.00058715</td>
</tr>
<tr>
<td>1000</td>
<td>0.00057854</td>
<td>0.00060339</td>
</tr>
<tr>
<td>2500</td>
<td>0.00031080</td>
<td>0.00063090</td>
</tr>
<tr>
<td>5000</td>
<td>0.00036976</td>
<td>0.00068166</td>
</tr>
<tr>
<td>7500</td>
<td>0.00069787</td>
<td>0.00073163</td>
</tr>
<tr>
<td>10000</td>
<td>0.00074707</td>
<td>0.00079670</td>
</tr>
<tr>
<td>12500</td>
<td>0.00081374</td>
<td>0.00086620</td>
</tr>
<tr>
<td>14285</td>
<td>0.00085234</td>
<td>0.00093793</td>
</tr>
<tr>
<td>20000</td>
<td>0.00124988</td>
<td>N/A</td>
</tr>
</tbody>
</table>

A.6 Distributed Chat

A.6.1 AiR Implementation

Listing A.4: AiR - Chat Client

```ruby
require File.dirname(__FILE__) + "../../lib/air.rb"
require File.dirname(__FILE__) + "/chat_server.rb"

class ChatClient < Actor
  def init(username)
    @username = username
    @chat_server = ChatServer.lookup("chat_server")
    @chat_server.>.register_client(actor_self)
  end
end
```
Listing A.5: AiR - Chat Server

```ruby
require File.dirname(__FILE__) + '/../lib/air.rb'
require File.dirname(__FILE__) + '/chat_client.rb'

class ChatServer < Actor
  def init
    puts "Initializing chat server..
    @clients = []
  end

  def register_client(client)
    puts "Registering client:
    puts client.inspect
    @clients << client
    puts "Client registered"
  end

  def broadcast(name, text)
    puts "Broadcasting message: #{name}, #{text}"
    @clients.each { |client| client.print_message(name, text) }
  end

  Air.start :name_server => "localhost:7777" do
    chat_server = ChatServer.spawn
    chat_server.register("chat_server")
  end
end
```

A.6.2 Java Implementation

Note: Customized version of code from a tutorial located at: http://pirate.shu.edu/~wachsmut/Teaching/CSAS2214/Virtual/Lectures/chat-client-server.html
import java.io.*;
import java.net.*;
public class ChatClient implements Runnable{
    static Socket clientSocket = null;
    static PrintStream os = null;
    static DataInputStream is = null;
    static BufferedReader inputLine = null;
    public static void main(String[] args) throws Exception {
        int port_number=2222;
        String host="localhost";
        clientSocket = new Socket(host, port_number);
        inputLine = new BufferedReader(new InputStreamReader(System.in));
        os = new PrintStream(clientSocket.getOutputStream());
        is = new DataInputStream(clientSocket.getInputStream());
        if (clientSocket != null && os != null && is != null) {
            new Thread(new ChatClient()).start();
            os.println(args[0]); // set user name
            while (true) {
                os.println(inputLine.readLine());
            }
        }
    }
    public void run() {
        String responseLine;
        try{
            while (!(responseLine = is.readLine()) != null) {
                System.out.println(responseLine);
            }
        } catch (IOException e) {} 
    }
}

Listing A.7: Java - Chat Server
import java.io.*;
import java.net.*;
import java.util.Vector;
public class ChatServer{
    static Socket clientSocket = null;
    static ServerSocket serverSocket = null;
    private Vector<clientThread> clients;
    public static void main(String args[]) throws Exception {
    }
```java
int port_number=2222;
serverSocket = new ServerSocket(port_number);
}

public ChatServer() {
    System.out.println("Initializing chat server");
    clients = new Vector<clientThread>();
}

public void mainLoop() throws Exception {
    while(true){
        clientSocket = serverSocket.accept();
        System.out.println("Registering client");
        clientThread client = new clientThread(clientSocket, this);
        client.start();
        clients.add(client);
        System.out.println("Client Registered");
    }
}

public synchronized void broadcast(String user, String text) {
    for (clientThread client : clients)
    {
        client.os.println(user + " says: " + text);
    }
}

class clientThread extends Thread{
    DataInputStream is = null;
    PrintStream os = null;
    Socket clientSocket = null;
    ChatServer server;

    public clientThread(Socket clientSocket, ChatServer server){
        this.server = server;
        this.clientSocket = clientSocket;
    }

    public void run()
    {
        String line;
        String name;

        try{
            is = new DataInputStream(clientSocket.getInputStream());
            os = new PrintStream(clientSocket.getOutputStream());
            name = is.readLine();
            while (true) {
                line = is.readLine();
                server.broadcast(name, line);
            }
        }
    }
```
A.7 Contact Details

If you found the project interesting, or otherwise have any comments, please feel free to email me at ahj04[@]doc.ic.ac.uk.