Implementing Stage: the Actor based language

Final Report - June 2007

John Ayres

Supervisor
Professor Susan Eisenbach

Second marker
Dr. Tony Field

Imperial College London

Abstract

Intel's 80 core processor represents the state of the art in processor technology and to take advantage of these additional cores programmers will have to write concurrent code. Writing massively concurrent code in existing languages to achieve even modest concurrency is difficult. And then we have the challenge of getting the most out of commodity clusters of networked computers.

Programmers hoping to exploit these trends using existing mainstream languages and paradigms often fail because these languages do not fuse well with concurrency and distribution. The lock-based concurrency of many existing languages including JAVA and PYTHON is clumsy and error prone and many languages sorely lack high level distribution semantics. The Actor model of Hewitt, Bishop and Steiger is a model which fits more naturally with concurrency and distribution. The model exudes concurrency and by prohibiting shared mutable state the model prevents many of the errors associated with lock-based synchronisation.

Early Actor-based languages enjoyed only moderate success, probably because they were before their time. More recent Actor languages have enjoyed greater success, the most successful being ERLANG, but the language's focus is still to narrow and its syntax too esoteric to be of general use. Other existing languages suffer from obscure syntax, immaturity, poor scalability and poor performance. There is a need for a language which presents a usable, general and fast encoding of the Actor model.

We present STAGE, our new Actor-based language, with distribution, concurrency and process mobility at its core. The language is embedded in PYTHON and inherits PYTHON's lightweight syntax and flexibility which is coupled with the full power of the Actor model. We developed several large programs to demonstrate its expressiveness and performance. We are pleased that it substantially outperforms the JAVA based Actor language SALSA.

Acknowledgements

I would like to thank Professor Susan Eisenbach, Dr. Tony Field and Matthew Sackman for their support, ideas and encouragement. Many thanks to Neil Dunn and Richard Hayden for their insightful and sometimes off-topic conversations. Thanks also to my parents, whose continued support has got me this far. Finally I would like to thank my flatmates James Dicken and Harry Tabner for putting up with me.

Contents

List of Programs

Chapter 1

Introduction & background

The current trend in CPU design is towards multiple core architectures. To take advantage of such architectures programmers must carefully spilt their programs into components which can be executed in parallel. But the dangers of optimisation are well known. W.A. Wulf notes,

> *'More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.'* [80]

It is widely accepted that programming in concurrent environments is hard. Concurrent applications are vulnerable to a set errors not present in single threaded code. In languages that permit sharing of data between threads, protecting data from low level corruption caused by concurrent access is error prone. Using existing mechanisms, ensuring that threads interact consistently is challenging.

Applications resident on a single host are constrained by the performance, capabilities and connectivity of the host. To increase performance applications may requisition additional hosts to provide further scope for parallisation. In other applications distribution is essential to provide access to information which is only available remotely, or to gather information from remote locations. Prominent examples include web services and remote data access. Google famously use of a cluster unreliable, low-end, commodity hardware to create a logical high performance, reliable and scalable computing cluster [15]. Can we create systems that exploit connected collections of computers? How easy are such systems to build, modify and maintain?

The advent of mobile and pervasive computing has given rise to a new set of operating conditions. Humans are carrying an increasing number of mobile devices from laptops to mobile phones and PDAs. The June release of the Apple iPhone, which features a cut-down version of the Mac OS X Operating System with built-in WI-FI, demonstrates the increasing power of modern mobile devices. These devices are synonymous with low bandwidth communication operating over intermittent connection mediums. Furthermore communication may have an enforced monetary cost or may deplete valuable mobile resources including battery life and processor time. Devices may join and leave the network at any time either through explicit action or unexpected loss of connection. Such networks may not be complete - each host may

only be capable of communicating with a few of its closest or trusted hosts. This environment renders the client-server model unsuitable since a fixed server is a central point of failure and a performance bottleneck. On these devices execution must proceed independently and autonomously.

Process mobility allows processes to execute independently of their underlying hosts. Mobile processes can move freely about the network taking with them data they have gathered or partial results generated by long-running computation. Mobile processes can use their migration capability to achieve a number of goals. Firstly to maximise performance by moving to a higher performance or more suitable host. Such processes may reduce their overall runtime and network message volume by moving closer to a resource or comrade. Other important uses include load balancing and long-life whereby processes migrate to escape failing hosts. From a user's perspective mobility allows user-visible applications such as email clients and web browsers to move between hosts. A motivating example is that of a businessman who begins composing an email on his desktop computer, at the click of a button the application moves to his mobile device. He finishes and sends his email during his commute to work.

The Actor model of Hewitt, Bishop and Steiger [41] is a model where the underlying behaviour is that of independent execution and asynchronous communication. Actors are active entities and interact by exchanging messages. Actors are a more natural construct for dealing with concurrency and distribution. Actors communicate via message passing and they do not share state. The lack of shared mutable state means that Actors are not subject to the low-level corruption prevalent in many popular languages. The Actor model can be extended to allow such messages to span hosts on a network which supports inter-host communication and distribution. Actors encapsulate both data and behaviour and can be extended to support process mobility.

Existing Actor-based languages often seek out a particular niche. SALSA [74] favours distribution at the cost of local speed, SCALA event-based Actors [37] favour lightweight switching and supports a high volume of Actors, and ERLANG [13] favours reliability and development time. Whilst these languages are successful in achieving their aims they are often too specialised or confusing to be of general use. This project creates a language which presents abstractions that make the Actor model more consistent with current Object Orientated methodologies.

1.1 Contributions

This project presents a new Actor language, named STAGE [1] which supports concurrency, distribution and process mobility. The need for concurrency and the dangers associated with concurrent code are presented in chapter 3.

The development of the language was both incremental and reciprocal - the language is used to develop applications and development experiences shape the language. The language takes PYTHON, a dynamic Object Orientated language, and extends and modifies the existing language constructs to provide a new Actor language which combines the lightweight syntax of PYTHON with powerful Actor-language abstractions. The language is used as a basis for an evaluation of the Actor-model. I evaluate the extent to which the Actor model is suitable for

[1] The name STAGE was chosen by Neil Dunn.

real-world application.

In chapter 5 I present solutions to the problems encountered during the development of the language including distributed naming and rendezvous, dynamic networks, interaction with the Operating System and load balancing. I evaluate STAGE's primitives, identify a weakness and implement 'lazy synchronisation', a concept based on Lieberman's futures [53], which removes explicit synchronisation. Those wishing to understand STAGE's constructs or view typical STAGE programs should proceed straight to this chapter.

In chapter 6 I show how STAGE programs are smaller, simpler and more intuitive than other languages. I demonstrate how distributed systems can be built quickly and without extra components or frameworks - a problem prevalent in JAVA. To demonstrate that STAGE is suitable for the development of large real-world applications I present the following applications:

- A multiplayer distributed realtime graphical game.
- A distributed web server and mobile browser.
- An efficient flexible network monitoring tool based on process migration.

In the same chapter I present quantitative performance measurements that demonstrate that the language is capable of increasing performance in multiple core and distributed contexts. I show that STAGE significantly out performs SALSA (a state of the art Actor-based language). I show that load balancing can be successfully applied to a population of Actors to optimise performance. Finally I show how STAGE's threading model can support in excess of 14,000 Actors without significant messaging slowdown.

In chapter 2 I give a more detailed overview of the Actor model. I discuss how the model can be extended to support distribution and mobility. I present a selection of constructs which are found in existing Actors languages.

In chapter 4 I investigate how a popular mainstream language, JAVA, handles distribution and concurrency. I present four state of the art Actor languages and evaluate the usability and style of each of these languages.

Chapter 7 outlines the lessons learnt during development. I present a selection of features and improvements which could be made to enhance the language.

Chapter 2

Actors

2.1 The Actor model

The language developed during this project, STAGE, is based on Hewitt's Actor model. This chapter outlines the pure Actor model [41, 8, 10] such that readers may familiarise themselves with what is a simple, yet powerful model of concurrent computation. An appreciation of the underlying model will provide context to the STAGE language and gives a common starting point from which to evaluate existing Actor-based languages. A solid background will allow us to identify and evaluate features in STAGE which deviate significantly from the formal definition of the Actor model. The section concludes with a cross section of useful techniques, abstractions and constructs which various Actor language researchers have proposed. Many of the features presented have positively influenced the development of STAGE.

2.1.1 What is an Actor?

> "The Actor metaphor for problem solving is a large human scientific society: each Actor is a scientist. Each has her own duties, specialties, and contracts."[43]

We can view a programmatic Actor as a simplification of its human counterpart. Each (human) Actor has a number of acquaintances (other Actors with whom he is familiar) with which he may choose to communicate. When spoken to by a fellow Actor an Actor exhibits some behaviour - this may be to ignore the conversation or to take some action. The computational Actor model uses these two aspects: behaviour and acquaintances to describe Actors of arbitrary complexity. I have not defined 'behaviour' precisely since doing so appears, at first, to trivialise the model. Readers should consider the Actor model as an 'assembly language for concurrency' with the enormous potential for higher-level abstractions. To give consistency to Actor implementations an Actor's behaviour is defined to be a selection of the following operations [8]:

- Sending a finite number of messages to other Actors (`send`).

- Specifying a replacement behaviour (`become`).
- Creating a finite number of new Actors (`create`).

The parenthesised text in the `Typewriter` typeface at the end of each operation's definition is the keyword most commonly used to describe the behaviour in abstract and concrete implementations of the Actor model. For brevity the specification of an Actor's behaviour often called an Actor's script [52]. Much like Object Orientated models, where we make the distinction between a class definition and an object instance, we make a similar distinction between an Actor instance and the script (definition) of an Actor. An Actor has one script yet many Actors share the same script. I use the term 'Actor' to refer to both the script and a specific Actor instance interchangeably. This is for clarity since in examples I will often give Actor definitions (scripts) and not explicitly create instances, the intended meaning will be clear from the context.

The model, as presented, is an abstract definition. Those more familiar with Object Orientated methodologies and languages (I include myself in this group) may prefer the less precise but more tangible definition that follows. Active objects [49] are objects which posses the ability to execute independently - conceptually a thread 'trapped' within an object. An Actor can be considered a constrained or restricted version of such objects. The restrictions are:

- All inter-Actor communication is by message passing and message passing alone.
- Actors may not share state. Any data sent in a message is a copied - no references to an Actor's internal state are permitted to leave an Actor.
- Actors may only communicate with acquaintances which they have successfully accrued during their lifetime.

The immediate benefit of such restrictions is that they prevent the variable corruption prevalent in many multi-threaded languages which allow sharing of state. I discuss the dangers inherent in multi-threaded code and the Actor model's susceptibility to such errors in section 3.2.

2.1.2 Modelling a scenario

> "A Hare one day ridiculed the short feet and slow pace of the Tortoise, who replied, laughing, "Though you be swift as the wind, I will beat you in a race." The Hare, believing her assertion to be simply impossible, assented to the proposal; and the Tortoise asked the Fox to choose the course and fix the goal."
>
> – A simplified excerpt from Aesop's 'The Tortoise and the Hare'.

At the Actor model's core is the concept of a role. A role is an active obligation to perform some action. I fall-short of describing this as a goal since goals are more typically associated with agents which are close relative of Actors but are usually considered higher-level. To solidify the concept we apply the Actor model to the excerpt above. To model a scenario or interaction using the Actor Model we must firstly determine [43]:

- The Actors of the system.

- Which messages each Actor should receive.
- The behaviour each of the Actors should exhibit upon receipt of a message.

We should also determine the acquaintances each Actor must possess: an Actor cannot interact directly with another Actor unless they are acquaintances.

When applying the methodology to the excerpt we designate the Tortoise, Hare and Fox as Actors in our system, although arguably the 'course' is also an Actor since the Actor model adopts the mantra 'Everything is an Actor'. Listing 1 shows how behaviours might be extracted from the text (where `receive A -> B` represents receiving message A and performing action B in response). Listing 2 defines the acquaintances each Actor must possess to fulfil its role. In

Listing 1 Behaviours in 'The Tortoise and the Hare'.

```
behaviour(Tortoise):
  receive ridicule -> send challenge to Hare
  receive acceptance -> send course request to Fox

behaviour(Hare):
  receive challenge -> send acceptance to Tortoise

behaviour(Fox):
  receive course request -> send course to Hare and Tortoise
```

Listing 2 Acquaintances in 'The Tortoise and the Hare'.

```
  acquaintances(Tortoise) = {Hare, Fox}
  acquaintances(Fox) = {Tortoise, Hare}
  acquaintances(Hare) = {Tortoise}
```

this acquaintance relation whilst the Hare is an acquaintance of the Fox the reverse is not true. The relation need not be symmetric: we favour each Actor having a small set of acquaintances with little symmetry since this represents low coupling which increases the potential for Actor reuse.

In the example the Hare cannot communicate directly with the Fox without first gaining Fox as an acquaintance (learning about the existence of another Actor is discussed in section 2.2.5). Hewitt describes arbitrary Actors A and B as being 'mutual acquaintances' when A is an acquaintance of B and B is an acquaintance of A. If we wish to model a call-and-return interaction between two Actors then we require that (potentially implicitly) the Actors are mutual acquaintances.

2.2 Actor language constructs

The previous section outlined the Actor model in abstract where Actors can receive messages, create Actors, and specify a replacement behaviour upon receipt of a message. Concrete imple-

mentations must deal with the specifics of this 'low level' model.

The abstract model does not feature many language constructs which programmers use everyday such as method call semantics and does not explicitly specify distribution nor is a concrete syntax provided. Thus there are a number of frequently performed tasks which would be both verbose and difficult to write when represented in an unembellished and faithful encoding of the Actor model such as SAL (Simple Actor Language) [8]. For this reason a number of extra features, libraries, extensions and tools have been suggested to aid the creation of Actor based systems. A number of the more prominent tools and concepts are presented.

2.2.1 Futures

Futures are presented in [40] although the report's focus is on the garbage collection of futures rather than the semantics and use of futures. Lieberman discusses the use of futures in the context of an Actor based language in [53].

Lieberman describes a future as an 'I.O.U' for computation. Essentially an Actor may return a future rather than a concrete and computed result. A future is an Actor like any other and has the potential to perform computation in parallel. One of the goals of STAGE is to promote parallelism at the language level; futures provide such parallelism making them a useful addition to the final language.

Futures allow Actors to remain responsive. A hypothetical 'Server' Actor's aim will be to remain responsive to requests by minimising the time it spends not checking its message queue. If, in response to a message, such an Actor performs computation on its own thread then, by definition, it is 'away' from its message queue. To avoid this the Server may instead create, configure and return a future which will service the request. The server may continue to service its message queue as soon as it has generated and returned a suitable future.

Futures also benefit the recipient by allowing the recipient to decide when (and if) they 'cash in' their 'I.O.U.' (the future) to determine the concrete result. A future usually provides methods to access the result synchronously and to determine whether the result has been computed yet (to allow asynchronous interaction). A possible extension is to allow the holder of a future to subscribe to the future to receive a notification when the result is 'ready'. A further advantage is that futures can be passed from Actor to Actor, allowing an Actor to simply forward the (potentially still to be computed) result of an invocation.

The prevalence of futures in a language may obscure an Actor's logic: a programmer may repeatedly create new futures simply to return results. It is my opinion that futures should be included in the underlying language (to implement various behaviours such as synchronous/asynchronous method invocation and to allow Actors to remain responsive) rather than as a pattern that an Actor programmer must explicitly code.

2.2.2 Mobility

An informal definition of code mobility is given in [32]: "[Mobility is] the capability to dynamically change the bindings between code fragments and the location where they are executed".

More informally mobile code is code which, during its execution, can migrate (synonyms: move and transfer) from one host to another. Such code resumes its execution on the destination host. We can make this definition more precise by considering resumption and transfer. Resumption, in the previous statement, can take one of two forms:

1. Execution continues from the point of migration.
2. Execution continues from some specified point.

Similarly transfer can be of either:

1. Code (often called a script) and data (the Actor's state).
2. Code, data and the state of execution (usually the stack).

From these axis we can define two mobility styles: weak and strong mobility [32]. Strong mobility provides resumption style (1) but requires transfer style (2). Weak mobility provides the weaker resumption style (2) but requires only the simpler transfer style (1). The benefit of strong mobility is that execution resumes from the point of migration, which makes it the more general style. Under this model Actors can be moved at any point in their execution at either their own request or at the request of another Actor. Strong mobility does not require code that explicitly deals with the resumption of an Actor on a new host. Many Actor/agent languages use this approach including, surprisingly, a module for the Linux kernel[1]. Weak mobility requires that Actors specify, or declare which code should resume the execution of the Actor on the destination host. In practice weak mobility is sufficiently expressive since, in the real world, when an Actor migrates it must take actions to acclimatise itself to its new environment. Actions include re-establishing contracts with local services in the new environment, locating collaborators and reopening files. In some sense the ideology of arbitrary subjective migration without an Actor's explicit knowledge associated with strong mobility is a misnomer. I make this statement after coding large examples where few Actors (although notably computation-bound Actors are amongst this few) execute completely independently of local resources.

Listing 3 Strong and weak mobility in pseudo-code.

a) Strong mobility

```
start:
  print 'On host A'
  migrate B
  print 'On host B'
```

b) Weak mobility

```
start:
  print 'On host A'
  migrate B

arrive:
  print 'On host B'
```

[1] Mobility for Linux, see http://lwn.net/Articles/223839/

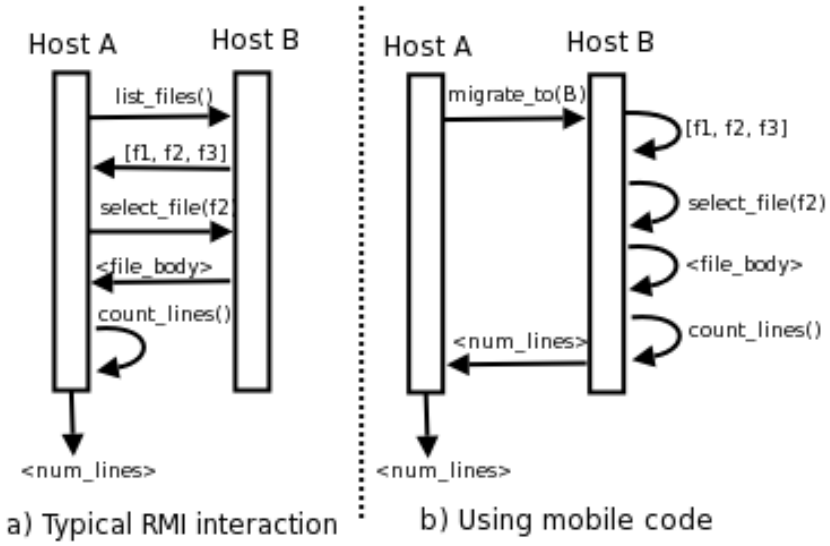

Figure 2.1: Reducing network traffic (with inspiration from [11]).

The mobility of code is not achieved without cost. Even under weak mobility Actors must still transfer their code and data. The inclusion of code mobility in a language must be justified. The motivating uses of mobile code follow:

Program mobility - The advent of mobile and pervasive devices has opened the possibility of inter-device migration of running programs. An email client that jumps from your desktop to your next generation mobile phone is a much exalted example but mobility can be put to more imaginative and pervasive uses. Imagine your eating habits migrating from your fridge, to your PDA and finally to your shopping trolley.

Maintenance of software - The IEEE defines software maintenance as "the modification of a software product after delivery in order to correct faults, to improve performance or other attributes, to adapt a product to a changed environment". Software maintenance is difficult problem for which there is no silver bullet. However Actors/agents can be used to modify, update and reconfigure running software [55].

Load balancing - If a host becomes overburdened with computation it can request the migration of resource-hungry code fragments. Whilst this may seem trivial, in reality load balancing of autonomous Actors is hard. Actors are often long running and require continuous balancing which precludes one-shot balance-compute-return techniques. Furthermore Actors may change their load profile at any point during their execution - they may exhibit cycles of high-load followed by relative inactivity. Load balancing is discussed further in section 5.3.8.

Low bandwidth/unreliable connections - Once an Actor has migrated to a new host it can continue its execution independently of its prior host. Consider a simplification of the control logic for the Mars Rover[2] which collects and sends data back to a base station. A naive implementation of the communication system might involve a persistent connection back to the base

[2]The Mars Rover - See http://marsrovers.nasa.gov/home/.

station along which any data collected is sent. However the Rover may go out of range or its communication link may be obstructed by a Martian rock resulting in data loss[3].

Fault tolerance - Collections of agents can migrate, replicate and spread themselves across a network reducing the reliance on any one node. This increases the fault-tolerance of a system as a whole. Actors can leave hosts which display signs of failure such as reduced bandwidth, high temperatures, low memory, and other indicators. This prolongs an Actor's lifespan which is essential if high reliability is required.

Continuous running - A host wishing to leave (possibly temporarily for maintenance, upgrade or replacement) a distributed system can migrate any system-critical Actors to others hosts before leaving the system. Without mobility systems must be explicitly designed to tolerate the temporary loss of a host or systems must provide bespoke mechanisms to transfer control to other hosts.

Locality and efficiency - Processes may migrate closer to the resources on which they operate or collaborators with whom they communicate. A process whose task is to perform a complex operation on remote data may migrate to the remote host to gain local access to the data (figure 2.1). DJ (a variation of JAVA RMI with mobility) has been used to optimise inter-host communication [11].

Intelligent workflow - This technique splits a business process into autonomous components whose goal is to complete the work associated with their role. As an example consider an email system for a bank whereby customers make requests. Requests might include queries such as 'How much money is in my account?', commands such as 'Send me a new debit card.' or any other reasonable request. If we represent the 'email request' as an Actor then the Actor's goal is to collaborate and distribute tasks to other Actors who are able to fulfil the request. To do this Actors may move around the system to find suitable collaborators using a work distribution system such as Contract Net [70].

2.2.3 Pattern matching

Pattern matching is a method for concisely declaring alternate behaviours based on the contents and/or structure of a message. For discussion we consider a 'post office process' which receives a message (representing a physical letter) containing the postage paid, the destination postcode and the body of the letter (as plain text). The rules of the post office are as follows:

- Letters with no postage paid (i.e. 0) should be returned to the sender.
- Letters with a postcode that begins 'CM' (Chelmsford area) should be delivered by hand.
- Finally all other letters should be forwarded to the national sorting office.

Using pattern matching specifying this behaviour is both clear and concise. The ERLANG declaration is given in listing 4. Without this facility we would have to resort to nested conditionals

[3]However utilising Dr. Vinton Cerf's Interplanetary Internet might make this a viable solution.

Listing 4 A post office in ERLANG.

```
receive
  {0, postcode, body} -> return_to_sender(body);
  {N, "CM" ++ code, body} -> deliver_locally(body);
  {N, postcode, body} -> forward_to_sorting_office(body);
end.
```

which would be both longer (with respect to lines of code) and would make the different behaviours harder to discern. A disadvantage of this approach is that it creates a dependency on the format and structure of a message rather than purely its contents.

2.2.4 Synchronous method invocation

There are a number of situations where a programmer may wish to invoke a (possibly remote) method and determine its result *before* any other operations take place. Consider a 'set top box' which allows a viewer to watch digital television. We assume that its role is to receive messages from the user specifying the channel the user wishes to view. Imagine a 'child protection' system which prevents children from watching the television after 9PM (possibly due to the proliferation of adult content). The `SetTopBox` Actor must (before answering any requests for channels) check the time. GLINT allows synchronous invocations and the solution in a GLINT inspired pseudo-code is shown in listing 5. The logic is clear: if the time is later than 9PM

Listing 5 Watching the television in pseudo-GLINT.

```
define SetTopBox(TimeServer timeServer)
  watch(Channel channel)
    if(timeServer.getTime() > time('9PM'))
      return BlankTVStream('ERROR: Too late for TV!')
    else
      return TVStream(channel)
```

then we return an error else we allow the user access to the channel. The same code in SALSA (listing 6) requires the declaration of a 'helper Actor' and is much less concise.

2.2.5 Naming

To send a message, and thus interact, with an Actor in the pure Actor model requires that the target Actor be an acquaintance of the Actor originating the communication. 'Becoming acquainted' with Actor can be accomplished two ways. Through creation - if Actor A creates Actor B then A learns the name of B. Through messaging - Actor A may learn the name of Actor B by receiving a message containing the name of B. This can lead to an unnatural programming style; some Actors will have one instance (singletons) and can be referred to by their universally known name. A `Clock` Actor is an example of a singleton. Furthermore the pure Actor model does not discuss naming within a network: how does an Actor begin an interaction with another remote Actor?

Built-in Actors are primitive Actors which 'bottom out' a computation: preventing infinite

Listing 6 Watching the television in SALSA.

```
behaviour SetTopBox {

  SetTopBox(TimeServer timeServer) {
    this.timeServer = timeServer;
  }

  TVStream watch(Channel channel) {
    Decision decision = new Decision(channel);
    time <- getTime() @ decision <- makeDesision(token) @ currentContinuation;
  }
}

behaviour Decision {

  Channel channel;

  Decision(Channel channel) {
    this.channel = channel;
  }

  TVStream makeDecision(Time time) {
    if(timeServer.getTime() > new Time('9PM')) {
       return new BlankTVStream("Too late for TV!");
    }
     else {
       return new TVStream(channel)
    }
  }
}
```

regress [43, 8]. They are the Actor-equivalent of the primitive data types of other languages. The granularity of such built-in Actors is a design decision. One might consider a string a built-in Actor (much like the immutable strings of JAVA) or one may designate a character as a built-in and consider strings as lists of such characters (as found in C and HASKELL).

Sackman provides singleton Actors for GLINT [67] which are uninstantiable: there is one implicit instance per host. Singleton Actors may be referenced by name and specify no instantiation parameters since they cannot be instantiated by other Actors. These are similar yet distinct from built-in Actors since primitive Actors are instantiable (we may, for example, create a new string Actor).

SALSA drifts further from the pure Actor model by introducing the concept of a globally unique Actor name (UAN) which is supported by a universal Actor locator (UAL). This allows Actors to be located (and therefore communicated with) within a network of Actors.

2.3 Open systems

In [42] the authors describe open systems as 'open ended, incremental and undergoing continual evolution'. They present an example which describes the process of locating a fridge in 'good condition'. The example was selected to highlight two problems: large search spaces with undefined boundaries and informally defined (fuzzy) goals.

In general an Actor may have an intimate knowledge of the state of its current location and may

conceivably, through collaboration, have some information about Actors in other locations. But in an open system there can be an arbitrarily large number of locations and more locations (and therefore Actors) can join the system at any time. To paraphrase: no individual Actor can know precisely the boundaries and capabilities of the system in which it exists nor can it expect another Actor to possess such knowledge.

Consider a query to a system which has a well defined and verifiable objective such as 'find the telephone number of my local bank'. Assuming a starting point for the search, such a directory service, finding the information is relatively simple and, because we can verify the number (presumably by ringing it) we can be sure we have the 'correct' information.

If verification of the information is not available or the question is ill posed then we can determine what a certain Actor (the directory service) thinks is the correct information. If we ask multiple Actors and they present potentially conflicting information then we must resort to heuristics such as voting, trust, and accuracy metrics that we may have derived in previous communications (discussed further in [69]).

If STAGE is to produce such open systems then we must not introduce any constructs, assumptions or features which restrict the ability to form such systems. It would be tempting to introduce a global registry of Actors to aid interaction within the system. However such a registry would be both an obvious point of failure, performance degradation and would represent global state which we wish to avoid. A compromise is to have smaller-scale collections of Actors known as ActorSpaces [9] where name-based communication between mobile Actors within an ActorSpace is supported. This similar to the provision for broadcasting information within a subnet in traditional networking.

2.4 Conclusion

The Actor model is a simple yet powerful paradigm for concurrent problem solving. It provides a consistent protocol for communication amongst interacting components which execute concurrently. By embellishing the model with distribution and mobility constructs the model becomes a powerful abstraction for developing distributed systems. The Actor model prohibits sharing of state, so called 'unshared mutable state'. This prevents many of the errors and dangers associated with concurrent programming with shared state. I feel that the Actor model is quite low-level, the minimal demonstration Actor-language SAL, presented in [8], reveals that without higher-level abstractions the Actor model is not suitable for real-world application.

Chapter 3

Concurrency & distribution

3.1 Concurrency

3.1.1 Why concurrency?

As will be discussed shortly concurrent (multithreaded) code is an order of magnitude more difficult to write, to write correctly and to detect and fix errors than its single threaded counterpart. So why does a programmer resort to multithreading?

Firstly to overcome the limitations of a single thread of control. Informally with only a single thread of control at its disposal an application can only 'do one thing at a time'. Generally this is not a limitation since in single core CPUs there is only ever, logically at least, one operation occurring at any instant. The illusion of simultaneous execution is provided by time slicing whereby each process is given a short period (time slice) on the processor. Analogously with a sufficiently rich API it is possible to implement multithreading within a single Operating System process. In fact this is a technique known as user-space threading or a 'M x 1' model [7]. In this model one process has many threads. As a result each thread is extremely lightweight as switching, creating and destroying threads can be performed in user-space which avoids the necessity to trap into kernel space which is a costly operation. Actor-based languages such as ERLANG and SCALA use lightweight user-space scheduling to achieve the aforementioned benefits. 'Single threading' or a 'single threaded application', for this section at least, should be considered an application which is in possession of a single Operating System (or underlying virtual machine) thread. User-space threading is excluded from this definition for simplicity.

In which situations are single threaded systems not suitable? When the problem dictates that multiple actions are to occur at the same time. Mundane situations such as processing two data streams and more interesting situations such as flight control systems which continually monitor thousands of inputs and make thousands of decisions every second both require that their sub-actions can be executed concurrently. On a related note multiple users and interactivity present a similar problem for a single threaded application. If an application has the use of one thread to service N users then each user must wait for the thread to become free before

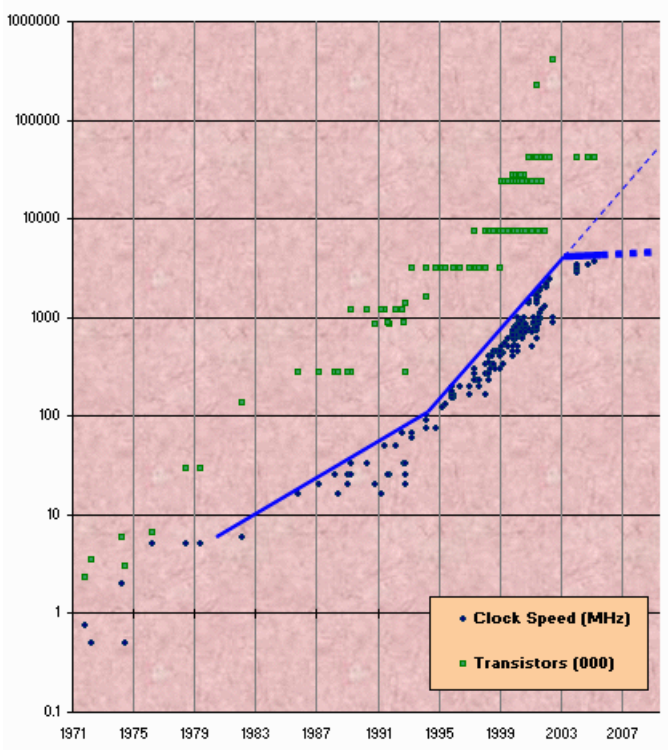

Figure 3.1: Transistor counts and clock speeds [71].

their request can be serviced. Clearly this is unsuitable in situations when interactivity or short (average per user) response times are required (e.g. the '10,000 user problem' [45]).

Any blocking or waiting, such as calling a method on an API with a synchronous interface, can necessitate the use of an extra thread. Synchronous I/O calls block the calling thread until the requisite input is available. For keyboard input this will be when the user has finished typing, for remote input this may be when the network stack has collected the relevant number of packets. If a programmer makes synchronous calls in a single threaded system then the system can make no further progress until the call returns. Poorly written graphical user interfaces often display this behaviour by performing I/O on a thread which is also responsible for redrawing the GUI. If the I/O is slow (maybe the network is congested, or the user types slowly) then the whole UI appears to 'lock up' since it cannot be redrawn until the I/O has completed.

3.1.2 Threading for speed

The previous section dealt with need for concurrency to provide functionality. Yet threading can also provide performance benefits. On a single CPU threading allows the computational work of other threads to continue whilst a thread is blocked. There is, however, an emerging need for well written threaded code to exploit the current trend in CPU design.

"Multi-core systems will soon become ubiquitous. Application design and the right developer tools will be key elements to unlock the maximum system throughput and performance of these multi-core powerhouses."[47]

Figure 3.1 shows the slowdown in the growth of processor speeds. It is for this reason that chip manufacturers are moving away from single core architectures to multiple core architectures. Some of the more prominent reasons for this slowdown are given in [2] and [71]. These are summarised below:

- Doping - This is the process of increasing the charge a localized area of silicon can store. Packan [65] claims we are reaching the limit of what we can achieve using dopants.
- Quantum effects - As logic gates get smaller quantum effects start to alter the classical behaviour of the gates.
- Power - Faster clocks mean more current flows through the processor [23]. This equates to an increase in power consumption. This problem is particularly relevant to mobile devices which have small power sources.
- Heat - The extra current flowing through the processor inevitably means an increase in heat generated. Removing this excess heat is becoming more difficult. Failure to remove this heat often results in 'hot laptop syndrome'.
- Physical limits - in [54] the author argues that the more energy a logic gate has the faster it can perform. Thus the overall speed is limited by its energy which, for a computer of fixed mass, is proportional to the speed of light squared. [71] alludes to this stating: 'light isn't getting any faster'.
- Cost - The monetary and intellectual investment required to increase the speed of a single core CPU is rising super-linearly.

3.2 Challenges when doing more than one thing at once

> "The vast majority of programmers today don't grok [understand] concurrency, just as the vast majority of programmers 15 years ago didn't yet grok objects" [71]
>
> "Concurrency is fundamentally hard; avoid whenever possible." [64]

The following section discusses some of the errors, problems and points for consideration that must be addressed when designing multithreaded systems. This is not an exhaustive list errors rather a discussion of how features in existing languages can both avoid and (in some cases) encourage such errors. This section is included to allow evaluation of STAGE's potential for error - it is important that any additional constructs provided do not motivate poor design and misleading programs.

3.2.1 Deadlock

Deadlock (or to use Dijkstra's more emotive term 'deadly embrace') is characterised by a system 'stopping' or 'locking up'. In these scenarios the system can make no further progress. Deadlocks often occur when processes both hold exclusive locks on a resource and will not relinquish those locks until they gain further locks. Dijkstra/Hoare's 'dining philosophers' problem is an often cited example of a system with the potential for deadlock. The conditions required for a system to reach a deadlock were first stated by Coffman (1971) in [22] and are reproduced below:

1. **Mutual exclusion** - Tasks claim exclusive control of the resources they require.
2. **Wait for** - Tasks hold resources already allocated to them while waiting for additional resources.
3. **No preemption** - Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion.
4. **Circular wait** - A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain.

Any 'pure' Actor language is unable syntactically to generate a deadlock since the primitives provide only asynchronous communication. Such asynchronous communication prevents the 'waits for' condition (condition two). Since all conditions must be met for a deadlock to arise the prohibition of condition two prevents an Actor-based system reaching a deadlocked state. However it is possible to generate code which is semantically susceptible to deadlock: characterised by Actors which 'keep trying' to perform a task [8]. To 'keep trying' is to 'wait for' some resource to become available. This reinstates condition two, resulting in the potential for deadlock. Non-pure Actor languages which provide synchronous method invocations (call-and-return) are susceptible to deadlock at the syntax (low) level.

3.2.2 Livelock

Livelock is a close relative of deadlock. In a livelocked situation the system does not 'grind to a halt' rather only superficial (non-useful) progress is made. There are a plethora of examples however a popular[1] example is that of two people walking down a corridor. Both move to avoid each other but both continually move the same way. Thus even though both parties are in motion (superficial work) no useful work is performed (neither get any further down the corridor). Livelocks often occur when deadlock recovery systems resolve a potential deadlock by restarting a process and the system then reaches the same deadlock which causes recovery, which then reaches deadlock, and so forth. Possible solutions include random retry as used in the CSMA/CD (network) protocol [44].

[1] I have been unable to determine the original source of this example.

3.2.3 Race conditions

A race condition occurs when the overall result of a system is dependent on timing. For this reason they are amongst the hardest errors to detect - there may be only one scheduling or timing of events that causes the problem. Deadlocks can be (dubiously) detected using timing (if no progress in made in N minutes then there's *probably* a deadlock) or more accurately using a 'waits for' graph (although implementing such graphs for distributed systems is non-trivial). 'Racing' systems may simply produce incorrect results (which are hard to check) rather than an obvious behavioural change. It is difficult (potentially impossible without prohibitive restrictions) to prevent such errors at the language level. We should however ensure that our syntax is clear and that subtle omissions do not give rise to these errors. In section 4.3.1 we discuss how SALSA (an Actor-based language) mixes imperative JAVA with token passing continuations and has the potential to create correct 'looking', but fundamentally *wrong* code.

3.2.4 Starvation and fairness

This is an often ignored but nonetheless important consideration. Starvation occurs when a process cannot gain access to the resources it requires. A common cause is priority. Processes with lower priorities may wait behind an endless stream of higher priority processes. In such situations the lower priority processes are starved. When considering priorities we should not restrict our thoughts to explicit programmed priorities observed by a scheduler (which are usually an integer value). Priorities can be derived implicitly from attributes, timing (when the request occurred) or the current state of the resource. An example whereby the priority of the client is based on the state of the resource and an attribute of the client is a single lane bridge where cars coming from a particular direction prevent cars from the opposite direction accessing the bridge [56].

In the Actor model Actors receive requests as messages in a queue or 'mailbox'. Languages such as ERLANG and QUPROLOG allow introspection of the queue through pattern matching: if a pattern doesn't match a message then it may sit in the queue forever. This may arise when an Actor modifies its behaviour to no longer accept messages of a type that it previously accepted. Actors whose messages were previously accepted and processed will now have their messages sitting in the target Actor's mailbox. This can lead to starvation of the old clients. Languages in which an Actor is forced to process messages in the order they arrived means that an Actor is encouraged to acknowledge all messages, although an Actor may still choose to simply ignore (perform no operation upon receipt of) certain messages.

3.2.5 Corruption, integrity and atomicity

Many languages allow memory to be shared amongst multiple threads or processes (including C, C++, JAVA and PYTHON). This paradigm is termed 'shared mutable state'. In these languages it is possible for multiple threads to modify a variable concurrently. If the operation is not atomic then low level corruption can occur. An atomic operation is an operation that appears to occur as one indivisible operation. Operations which are syntactically one token (incrementation, assignment etc) are often multiple operations in the underlying machine. A

simple operation such as `a=a+b` may result in the following (stack based) pseudo-assembly code (listing 7).

Listing 7 Adding two numbers using a stack.

```
1 PUSH a
2 PUSH b
3 ADD
4 POP a
```

If two threads perform this operation then the result is non-deterministic: 'a' can take the value $a + b + b$ (the 'correct' value) or $a + b$ depending on the interleaving of the operations. Doug Lea provides the `java.util.concurrent.atomic` package for JAVA which provides atomic versions of many common variable types.

With more complex data types such as strings (e.g. C-style character arrays) these interleavings can leave the data type inconsistent: the loss of a string's termination character can cause catastrophic behaviour (usually invalid memory accesses) if the string is passed to any of the standard string manipulation libraries. Languages which permit shared mutable state must necessarily provide a tool-set for preventing such errors. Commonly found tools in such a tool-set include locks, semaphores and monitors. Composition of objects that contain synchronisation operations can lead to deadlock or unintended behaviour. The problems and solutions involved in composing such code is a complex research area which I will not discuss, as motivation readers should consider whether composing two thread safe data structures results in a thread-safe composed structure.

Actor based languages avoid low-level corruption as only one thread (the Actor's thread) may access an Actor's internal variables. Any transmission or passing of variables is performed by copying and messaging. Higher level integrity constraints are still problematic. Consider a set of 'bank account' Actors which each represent a bank account belonging to a customer. If we sum the balances of all the accounts whilst a 'bank clerk' Actor is moving money from one account to another then we may see an under or over count of the total money held by the bank.

3.3 Canonical concurrent & distributed examples

To develop a language without reference to typical usage is unwise. To test the language we select a set of well defined examples. These are implemented during development and guide, motivate and select features for inclusion in the language. During evaluation these examples allow a comparison of the conciseness, elegance and expressiveness of STAGE versus other languages. The documentation for many existing languages includes only the examples which fit well with the language. The following examples have been generated to ensure that STAGE does not focus on a particular niche. For each of the following examples many conflicting definitions exist, but for the purposes of this project the following definitions will be considered authoritative.

3.3.1 Asynchronous countdown latch

An asynchronous countdown latch (or ACL) is parameterised by an initial countdown value. Each invocation to the `join` method decrements the internal count value by one. When the count reaches zero all Actors which have previously called join are notified (by callback). Doug Lea's `util.concurrent` package [51] contains a `CountDownLatch`, although this has blocking semantics. During my research I implemented this pattern in a number of languages. Implementing an example, even one as small as this, highlighted the massive variance in style and length that languages display.

3.3.2 Ping pong

In this example two processes swap `ping` and `pong` messages. The Gnutella protocol [35] uses a ping pong protocol to locate resources within the network. This example will send messages ad infinitum and each process will keep a count of the number of ping or pong messages it has received. A pseudo-code implementation is given in listing 8.

Listing 8 A pseudo-code implementation of the ping pong example.

```
1  PROC a:
2    while(1)
3       send ping to b
4       receive pong
5       count++
6
7  PROC b:
8    while(1)
9       receive ping
10      count++
11      send pong to a
```

3.3.3 Work distribution

This example concerns the distribution of work across multiple units. Each of these units can be placed onto separate (local) threads to take advantage of a potentially multiple core architecture. It should be possible to distribute these units of work further by involving networked nodes in the computation. It is important that the programmer be unburdened from the specifics of thread creation and networked distribution. This pattern is found in distributed computing initiatives such as the popular Seti@Home project [46]. An idealised pseudo-code for this example is given in listing 9.

3.3.4 Client server

In this model N clients communicate with one server. One of the most popular and widely known protocols that utilises this pattern is the HTTP Protocol. In listing 10 we have assumed

Listing 9 Distributed marking.

```
PROC lecturer(exam_papers):
  create markers

  while there are exam_papers left:
    for each marker:
      pass exam paper to marker

   wait until marks are returned

PROC marker:
  receive exam_paper
  migrate to a node in the universe
  mark the paper
  return the mark
```

Listing 10 A client-server interaction.

```
PROC client:
   send request to server
   wait until response

PROC server:
   receive request
   spawn worker thread to service request
     worker thread services request
```

blocking I/O semantics. It is for this reason that we must perform I/O (or any other blocking operation) using a different thread: if it is performed in the server's thread then other requests will not be serviced until the previous request completes. If our I/O routines have a non-blocking semantic, such as those found in the JAVA NIO API [59], then these extra worker threads are not required.

3.3.5 Bounded buffer

In this example we consider two processes: a consumer and a producer. In the general case we may have an arbitrary number of producers and consumers. An often quoted 'real word' example is that of a Burger Bar (inspiration from [57]). In the Burger Bar 'chefs' (the producers) grill burgers and place them on a heated rack. The rack has space for exactly 10 burgers. The customers (quite literally: the consumers) remove burgers from the rack as their hunger dictates. Even in this seemly simple interaction there are potential errors: consistency (ensuring each slot has a maximum of one burger and that two customers don't try to remove the same burger) and interaction (ensuring that no burgers are added when the rack is full and that no burgers are removed when the rack is empty).

3.3.6 Conclusion

Concurrency is useful and often essential. But by adding concurrency a programmer exposes his application to a range of problems not present in sequential code. Similarly detecting an error in sequential code is relatively straightforward - programmers supply applications with

test data and verify that the output is as expected. In concurrent code errors such as low-level variable corruption may appear infrequently and attaching a debugger to diagnose the problem may change the program's behaviour. This behavioural change may cause the error to disappear, such errors are often termed 'heisenbugs'. Components can interact independently and form systems rather than that input-output based programs. This makes testing harder since we must specify the expected behaviour and the functionally.

For these reasons writing concurrent code requires a different mindset and does not fuse well with existing languages. The barrier to entry is high - programmers who write good concurrent code using the correct tools are in the minority. Consistency is the best way to ensure error free code. Languages which allow a mixture of synchronisation techniques are susceptible to programmer confusion which ultimately results in error. Concurrency Orientated languages are essential.

Chapter 4

Languages

The following chapter outlines a selection of languages which, to greater and lesser extents, provide support for concurrency and distribution. These languages are presented to enumerate and evaluate salient Actor-based features, analyses of features not directly related to the Actor model are not provided. Languages where no implementation is provided such as SAL [8] (Simple Actor Language) and languages whose features are similar to those presented (notably PYTHON, C++ and C) are omitted.

4.1 Java

JAVA's design aims to allow development of systems 'that would let us do a large, distributed, heterogeneous network of consumer electronic devices all talking to each other.' [61]. JAVA is an Object Orientated language, although it is not a *pure* Object Orientated language due to its inclusion of primitives (int, boolean, etc.) which are not objects themselves, although their object encapsulated counterparts exist within the core `java.lang` package. Whilst we consider JAVA for the following section most of the arguments are applicable to other imperative languages including C and C++ although many of the synchronisation and threading operations are performed using libraries (such as the POSIX thread library - `pthread.h`) rather than as features of the language.

4.1.1 Concurrency

The `Thread` object is the unit of concurrency in JAVA. Each `Thread` has the potential to be run in parallel with other threads. Early versions of the JVM used green threading whereby threads are simulated by the JVM - the underlying Operating System has no knowledge of their existence. Most modern JVMs use native threads (Operating System threads) although mixed-model JVMs do exist. To execute code in the context of a new thread a JAVA programmer

must implement the `Runnable` interface[1] and then instantiate a new `Thread` object passing the `Runnable` object as a constructor parameter. Invoking the `start` method of the `Thread` begins the execution of the `run` method of the `Runnable` in the context of the newly-created thread. This 'library based' concurrency is discussed in [18], although the authors choose a variety of languages JAVA is not discussed in detail. Listing 11 shows the creation a thread to ping a remote server (the network orientated code for the ping command is omitted). Programmers

Listing 11 Pinging on a new thread in JAVA.

```
1  class Pinger implements Runnable {
2
3    public static void main(String args[]) {
4      new Thread(new Pinger()).start();
5      /* Main thread continues */
6    }
7
8    public void run() {
9      while(true) {
10       ping("www.google.co.uk");
11     }
12   }
13 }
```

who have not dealt with concurrency in JAVA are often confused by the relationship between an object instance and a thread. The confusion arises due to the fact that in simple single threaded applications objects' methods will only (and can only) ever be executed by the initial 'main' thread (we ignore garbage collection for this discussion). It is for this reason that many inexperienced programmers make no distinction between thread and object: believing objects to be inherently 'active'.

The reality is that objects in most Object-Orientated languages (such as JAVA) are passive [6]. A hypothetical `TransactionManager` is unable to manage any transactions without the help of a thread. So whilst developers aim for object encapsulation by 'hiding all of the details of an object that do not contribute to its essential characteristics'[17] they do not encapsulate the thread(s) that give rise to an object's behaviour. If it for this reason that assumptions are often made about the context in which the object is to interact. This is a major source of error: do we assume that our object will only be used by one thread or do we protect our object against potential concurrent access: gaining the so-called 'thread-safe' status?

4.1.2 Doing things together in Java

This section outlines the various synchronisation primitives provided by JAVA. Each object has a lock associated with it [36]. Any thread that invokes a method (or block) marked with the `synchronized` keyword on an object must first gain the object's lock. This is the primary mechanism for protecting a shared resource since the lock provides mutual exclusion.

An important implementation detail is that the lock is per-thread not per-invocation [50]. This means that whilst a thread holds a lock on a particular object it may call any of the object's synchronized methods without blocking. This allows both recursion and callbacks as long as

[1] We may also extend Thread however this limits the scope for inheritance since JAVA objects may only extend one object.

the callback is executed on the originating thread. If a synchronized method on object `A` is called which causes object `B` to 'call back' `A` then the callback must be performed by the original thread. If B uses a different (possibly new) thread to perform the callback then it will not hold `A`'s lock. At this point this new thread must acquire the lock, which can lead to deadlock.

JAVA also provides the primitives `wait` and `notifyAll`[2]. These primitives allow us to control the behaviour of threads primarily by forming monitors.

It is common practice to refactor [29] code into design patterns to provide reusable, proven, tested and familiar abstractions. Yet code dealing with concurrent interactions is often written using ad hoc synchronization where required. Doug Lea provides the popular `java.util.concurrent` framework [51] which provides familiar abstractions such as Blocking Queues, Thread Pools, and thread safe datastructures. These thread safe library objects are beneficial since they are well tested, well known and they effectively pull application-visible locks up into the library code - away from the main application logic.

> "Because inheritance exposes a subclass to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation'". [33]

Inheritance leads to a set of problems, most prevalent in 'thread safe' code, known as the 'inheritance anomaly' [19, 60]. One particular anomaly known as the history anomaly is shown in listing 12. In this example (which represents keybuffer for a telephone) an emergency call can only be attempted after at least three '9's have been added to the buffer (i.e. '9' has been dialed 3 times). We assume that `NumberBuffer` has implementations for `get` and `put`. The anomaly occurs when we have to override `put` simply to impose a restriction based on the telephone's history. To demonstrate the enormous scope for error I have deliberately omitted the `synchronized` statement for the overridden `put` method (the method in the superclass has the synchronized keyword) which could lead to inconsistent behaviour.

Listing 12 Making a mistake in JAVA.

```
1  class Telephone extends NumberBuffer {
2
3    public void put(int val) {
4      if (val == 9) ninesSeen++;
5      else ninesSeen = 0;
6      super.put(val);
7    }
8
9    public synchronized void emergencyCall() throws IllegalStateException {
10     if (ninesSeen < 3) {
11       throw new IllegalStateExcpetion("Dial 3 nines first");
12     }
13     /* Emergency call proceeds */
14   }
15 }
```

[2]The primitive notify also exists however its existence is for efficiency rather than functionality.

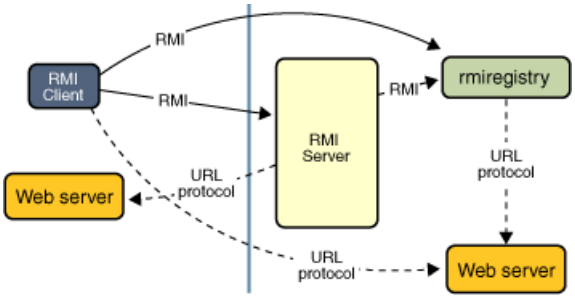

Figure 4.1: JAVA RMI and its components. © Sun Microsystems.

4.1.3 Distribution

Writing programs which interact across a network is a non-trivial task in JAVA. There are many ways achieve distribution including bespoke serialisation and socket communication, JAVA RMI, CORBA and SOAP messaging over HTTP. Each of these methods require the programmer to invest time in understanding these systems before implementation. This 'scaffolding' can take time to install and set-up, but I have often found that once the scaffolding is in place further distributed development is relatively straightforward. As outlined, JAVA provides language support for concurrency through the `synchronized`, `notifyAll` and `wait` statements but a programmer must select a library or framework to achieve distribution.

At this point the 'heavyweight' aspect of one of the most popular methods (JAVA RMI) should be noted. Whilst RMI allows us to execute methods on (potentially) remote objects whilst treating them as local objects this is not possible 'out of the box'. We must provide mechanisms (figure 4.1) for clients to obtain the body (bytecode) of 'mobile' objects and must provide interfaces for any remote objects. In large, continuously running, evolutionary systems these interfaces can limit flexibility. As Erich Gamma noted [75]:

> "In Java when you add a new method to an interface, you break all your clients. Since changing interfaces breaks clients you should consider them as immutable once you've published them."

4.1.4 Evaluation

JAVA, as demonstrated, has support for both concurrency and distribution. However the support is very 'low level'. JAVA is a verbose language. Much of this verbosity stems from its static type system (parodied in figure 4.2) and lack of support for common datatypes (lists, maps, trees, graphs etc.) at the language level. This verbosity makes determining the behaviour, locking and synchronisation patterns of concurrent code difficult. Furthermore JAVA developers are renowned for creating systems with multiple layers of indirection which further hides the *actual* behavioural code.

Figure 4.2: Verbosity in statically typed systems. © Gary Larson.

4.2 Erlang

ERLANG's development at Ericsson is introduced in [13] and is described as a language designed to support the development of large scale soft-realtime systems. ERLANG's design is influenced by developers' problem domain - telecoms systems. These systems have characteristics which include continuous (always running) operation, fault tolerance (if a deadline is missed the system should remain operational) and large scale distribution.

The development of the language was not a 'one-shot' up-front design rather it evolved with features being added and performance boosted as the requirements grew. The designers had a penchant for PROLOG and wanted a system with the features of PROLOG but with enhanced support for concurrency and error handling. The designers initially embedded their language in PROLOG, creating PROLOG meta-interpreter as described in [12]. The language developed into a fully-featured language, a language the authors describe as 'a strange mixture, with declarative features (inherited from PROLOG), multi-tasking and concurrency (inherited from EriPascal and Ada)'. ERLANG's current syntax borrows heavily from PROLOG.

4.2.1 Language features

Concurrency in ERLANG is initiated by the `spawn` primitive. The `spawn` primitive is passed an arbitrary function and arguments with which the process should begin its execution. The primitive returns a process identifier (PID) of the newly created process which can be used to

reference the process. Since PIDs are unique names for processes they can be considered a concrete implementation of Actor names. Processes (identified by their PID) can be sent a message using the `!` operator. Listing 13 spawns a new `Counter` process which increments its internal count variable each time it receives a message containing the atom `hello`. The process has been coded respond to a message containing an arbitrary PID and the atom `status` by sending a message containing its current state (its current count value, `N`) to the process identified by the PID it received. Lines 4 to 10 represent the definition of a counter process, although strictly

Listing 13 A 'hello' counter in ERLANG.

```
1  -module(count).
2  -export([counter/1, run/0]).
3
4  counter(N) ->
5    receive
6       hello ->
7         counter(N+1);
8       {Pid, status} ->
9          Pid ! N
10   end.
11
12 run() ->
13   Pid = spawn(count, counter, [0]),
14
15   Pid ! hello,
16   Pid ! goodbye,
17   Pid ! hello,
18
19   Pid ! {self(), status},
20   receive
21      N -> N
22   end.
```

it is a function definition. It is only the `spawn` command on line 13 that begins the execution of the count function concurrently. This provides flexibility as it is possible to define an Actor's behaviour by linking together a set of functions - we need not explicitly declare an 'Actor'.

Lines 6 and 8 contain patterns, these match the `hello` atom and the `{Pid, status}` pair respectively.

Lines 19 to 22 are used to query the counter. To query the counter a message is sent to the counter asynchronously followed by a blocking-wait for the counter's response. This allows a programmer to simulate a synchronous invocation on the counter. If the counter either fails to respond or responds with a message of a differing structure (which may happen if the counter's API is upgraded) then the callee will wait forever for the response (although it is possible to set a waiting time limit). For this reason the lack of support for request-response interaction as a language construct is a little disappointing but not a crippling omission.

Distribution was added to ERLANG in 1990. The syntax for `spawn` is enhanced to allow the creation of a process on a remote node. Listing 14 creates a process on node `Bob` which begins its execution by running the function `build`.

Listing 14 Spawning a process on a remote node in ERLANG.

```
1  spawn(build@Bob)
```

Distribution in ERLANG is very simple and as the language's target domain concerns high performance and reliable systems this is ideal since higher level distribution constructs come at the expense of increased complexity. From a functionality perspective it is possible to spawn processes on remote nodes, however further migration is not possible. In this sense all other nodes on the network are treated by the 'home' node as dumb slaves. Introducing processes at remote nodes and having them interact with other nodes in the network requires extra rendezvous and service discovery components. The language also provides the primitives `alive` and `not_alive` which test a node's willingness to be part of the computation. The usefulness of these operations must be questioned as `alive` simply checks whether a node is alive: it makes no contract with the caller that any subsequent remote spawn operations or messages sent will be accepted.

ERLANG also boasts 'runtime code replacement'. In a language designed for continuously operating telecoms systems being able to 'hot swap' code in and out (without restarting the server) is essential - nobody wants *their* call to be interrupted! Closer inspection reveals that this is simply a consequence of ERLANG's support for higher-order functions. The implementation given in [13] shows a server parameterised by a function which it applies to incoming messages. To change the behaviour of the server we send a message to the server containing the new behaviour. The server will use the function received in the message to process future requests. Note that a programmer must explicitly deal with code replacement in the logic of the code. This ability is not unique to ERLANG. Using this technique runtime code replacement can be implemented in an Object Orientated language by defining a sensible interface to a behaviour and applying the strategy design pattern [34].

4.2.2 Evaluation

> 'Certain applications cannot be efficiently programmed in ERLANG. We are considering adding imperative features to the language to solve these problems.'[13].

ERLANG contains a number of useful features and has an enormous affinity with the domain for which is designed. Yet it has a number of shortcomings. There is no notion of a process, thread or Actor in the syntax: processes are created by simply spawning a function. As the size of the program increases this may lead to unstructured code since determining which functions specify the behaviour of a process is difficult (impossible in some cases) through inspection of a process' source. ERLANG's impressive user-space threading model supports fast switching and creation of processes. Joe Armstrong (the inventor of ERLANG), demonstrates the power of ERLANG's lightweight threading model by setting and attempting the 'Armstrong challenge' [14]. The Armstrong challenge measures the scalability and high-load performance of an Actor language. ERLANG performs exceptionally well at this challenge. I use this challenge to evaluate my language in section 6.2.4. The initial implementation of the ERLANG virtual machine (on which ERLANG processes run) used one Operating System process and performed its own thread scheduling. Whilst this confers the benefits associated with lightweight threading it restricts the execution of ERLANG threads to one core of a potentially multiple core architecture. More recent releases create a process per CPU to increase performance on multiple core architectures.

ERLANG's pattern matching which is borrowed from PROLOG is a useful tool however complex patterns can make it hard to determine the interface (how other processes should interact with

a process) of a process without close examination of the `receive` primitive's set of patterns.

Though it is possible to write Object Orientated code in ERLANG through the use of records [66] objects are not a construct of the language. Sackman concurs, in [67] he notes '[there is a] lack of structuring features of the language'.

4.3 Salsa

SALSA [74] is programming language based on the Actor model. SALSA programs are pre-processed into JAVA before being compiled and executed as vanilla JAVA applications. SALSA programs are eventually compiled down to JAVA bytecode (class files) which allows SALSA programs to run on any machine running a JAVA Virtual Machine (the execution environment for JAVA). SALSA is discussed in detail as STAGE is embedded in PYTHON much like SALSA is embedded in JAVA. An analysis of SALSA highlights the pitfalls of integrating the Actor model into an existing language.

SALSA Actors can send messages, create new Actors and modify their own internal state. Sending messages and creating new Actors are both as set out in the specification of the pure Actor model (section 2.1) yet SALSA does not provide the ability for an Actor to *become* another Actor. In many Actor-based languages this is the method by which Actors specify a replacement behaviour. In SALSA an Actor changes its behaviour through modification of its internal state. The motivation for this approach is most likely because SALSA programs are translated into JAVA, an imperative programming language, where sequences of instructions are used to modify the state and thus the behaviour of an object.

Since becoming another Actor is a behavioural change it is possible achieve a result similar to the `become` keyword in SALSA through the use of the state design pattern ([33] p308) whereby we place a proxy Actor 'in front' of the actual Actor (which represents the current behaviour). The proxy Actor can then swap in and out behaviours as required. Morphable objects (such as the prototype implementation for SMALLTALK given in [79]) are another possible solution.

Many languages which feature the `become` keyword (including GLINT) and those which use recursion (such as ERLANG) insist that the entire state of the new process or Actor is passed to the new process or Actor. Consider a simple `Server` with variables `N` (the number of requests serviced), `name` (the name of the server) and `port` (the port number the server bind to). When a new request arrives the server must increment `N` since `N` is the request counter (we ignore the processing of the request for this example). In SALSA we can use the JAVA syntax for incrementation, namely `N++`. In ERLANG we are forced to specify the 'new' values of all the variables, even those that remain unchanged. This textual overhead is shown in listing 15. The new values for each of the variables are specified by position, which forces a programmer to remember

Listing 15 Passing state in ERLANG.

```
1  Server(name, port, N)
2    ...
3    Server(name, port, N+1)
4    ...
```

(or look up) the position of each of the variables in the process or Actor declaration (although good design practices may suggest that we encapsulate groups of variables into another Actor or object).

4.3.1 Message passing

Message passing in SALSA is represented by the <− token. This choice is a little confusing since ← is often used in pseudo-code to denote assignment. This is a better choice than '.', which is used in JAVA to invoke methods and access non-private variables, as these are synchronous operations - message sending in SALSA is asynchronous.

Throughout the following section we will use the example of a `Counter` Actor (declaration omitted) whose interface is made up of two operations: `reset` (which resets its value to 0) and `increment`. Listing 16 demonstrates how a programmer may (incorrectly) interact with

Listing 16 Interacting with a counter in SALSA.

```
1  Counter counter = new Counter();
2  counter <- increment();
3  counter <- reset();
```

a counter instance. Line 1 shows the creation of a new counter, we shall assume that the internal count variable is initialised to zero upon creation. This matches exactly the syntax for object instantiation in JAVA. Lines 2 and 3 show the sending of an increment message and a reset message. This example has been deliberately crafted to highlight how it is possible to write misleading programs. After execution of the above code snippet what is the value of the counter? In this example the value of the counter is non deterministic. It could take either the value 0 or 1. This is because SALSA makes no guarantee that the `Counter` will receive the messages in the order they were sent.

4.3.2 Token-passing continuations

Token passing allows us to impose an ordering on invocations and to pass the results of invocations between Actors. Listing 17 shows the use of a token passing continuation. The code invokes the `lookup` method on the `dns` Actor. The `dns` Actor then forwards

Listing 17 Using a token-passing continuation in SALSA.

```
1  dns <-lookup() @ webBrowser<-loadPage(token)
```

the result (as the token) to the `webBrowser` by invoking the `loadPage` method. We can use this feature to correct our earlier misleading code by replacing lines 2 and 3 with `counter <- increment() @ counter <- reset()`. We can create chains of arbitrary length using this approach although we must be careful not to specify the behaviour of the whole system in one Actor: a potential 'God' Actor.

4.3.3 Join continuations

Join continuations extend token passing continuations by allowing two (or more) messages to be sent whereby the next continuation in the chain (if any) is only invoked when all the results from the previous sends are received. Informally this allows us to invoke a number of methods on other Actors and then join the results together. If the `webBrowser` in listing 17 required not only the IP address (from the `dns` Actor) but also required the user's settings (from the `settings` Actor) then we can use a join continuation to perform both requests in parallel and then pass the result to the `webBrowser`. Listing 18 shows how one would modify listing 17 to support this functionality.

Listing 18 Using a join continuation in SALSA.

```
1 (dns <-lookup(), settings <- getSettings()) @ webBrowser<-loadPage(token)
```

Join continuations are also suited to performing work distribution. A programmer may spilt a (computationally intensive) task into subtasks then distribute portions of the task to a set of Actors. It then possible to use a join continuation to invoke the worker Actors and to retrieve the result when the whole task (i.e. all of the subtasks) has been completed.

4.3.4 Evaluation

One of SALSA's strongest features is that a programmer has access to JAVA's rich set of libraries. This means that SALSA can be used to perform 'real world' tasks rather than the small subset that most prototype languages can perform. That is not to say that most prototype Actor languages are not Turing complete, rather many lack some key features such as I/O capabilities, and a rich API that programmers have come to expect from production quality languages. The disadvantage is that there is not a comprehensive list of JAVA libraries and code that will interfere with 'correctness' of a SALSA program. The language does not warn nor prevent inappropriate interaction with JAVA. I believe, however, that if a programmer keeps the majority of the logic concerning concurrent interactions within SALSA's primitives then most mistakes can be avoided.

SALSA is unfortunate in that the current set of popular paradigms do not include token-passing and join continuations. Such constructs are not intuitive to most programmers although many would argue that they are more intuitive than chorded languages such as SCHOOL [26]. SALSA programs are aesthetically similar to JAVA programs and I can envisage many programmers making mistakes (listing 16) because the code looks like 'correct' vanilla JAVA. JAVA's rather verbose type system also hampers SALSA. Many short tasks such as reading a file require a large block of code. In larger examples such as trapezoidal approximation [1] the logic concerning the Actors' behaviours with respect to concurrent interactions is lost within the morass of imperative JAVA. Methods expecting the result of a join continuation must declare their argument as a primitive array type. If the method is expecting to be passed arguments of different types then the array must be typed: `Object[]` and the parameters removed from the array and cast to the correct type as required. Like many other Actor languages, support for synchronous method invocation is omitted. This does however allow SALSA to promote asynchronous communication and thus promotes responsive Actors.

The disadvantages of a rewriting strategy were highlighted when I used SALSA to contrast the performance of my own language. To determine SALSA's performance characteristics I had modify an existing SALSA implementation to make it more amenable to automated evaluation. Making a change to SALSA code involves (1) making the relevant change to the Actor code (2) translating the SALSA source into JAVA using the SALSA compiler (3) compiling the generated JAVA (4) running the JAVA bytecode. No tools are provided to automate this procedure and because the SALSA compiler rewrites into JAVA it does no error checking itself. It defers all checking, except simple syntax checking, to the JAVA compiler (javac). If an error is raised during the compilation of the generated JAVA a programmer is forced to sift through the esoteric generated code to determine the root cause of the error. Similarly ECLIPSE was unable to provide any syntax highlighting or support to aid the modification of SALSA source code. The results of the performance comparison are found in section 6.2.2.

4.4 Glint

GLINT is an Actor-based language which supports Actor mobility developed by Mathew Sackman as part of his master's thesis [67]. GLINT is a prototype language and as such has very few libraries and an immature syntax but demonstrates the features and extensions of a modern Actor language.

4.4.1 The GlintVM

The language sits on top of a custom virtual machine (VM), the so-called GLINTVM, which acts as an interpreter for the Safe Boxed Ambient calculus [58]. The Safe Boxed Ambient calculus is an extension of Milner's π-calculus with extra support for movement and locality. This VM provides an underlying abstraction that is closer to a mobile Actor model. The Safe Boxed Ambient calculus provides mobility primitives (allowing the encoding of mobile Actors) and makes no distinction between code and data. In a language which makes a distinction between code and data (such as JAVA) to achieve process migration one must distribute both the underlying code (or script) of the process and any data which determines the state of the process. To achieve strong mobility this may include an abstract representation of the program counter and potentially the state of the stack at the point of migration. Models which combine code and data do however suffer from a potential inefficiency. Consider a large number of Actors each with very little data but a large and complex task to perform on the data. If we distribute this vast number of Actors to a few hosts (to share the work) then, if we use a model where data and code are indistinguishable, we send the same 'code' (although we would not know it as code) multiple times. Sending code and data separately does mean Actors' scripts can be reused but this requires extra code and components and can lead to versioning and compatibility problems (as outlined, for JAVA, in [27]).

4.4.2 Distribution

GLINT supports arbitrary inter-host migration of Actors (listing 19). Migration in GLINT is sub-

Listing 19 Actor migration in GLINT.

```
1  define SomeActor(Number n)
2    ....
3      become AnotherActor(n.plus<One>) in host:anotherHost
4    ...
```

ject to a number of restrictions. The first restriction is that an Actor must lose any acquaintances it may have gained on its local host upon migration to a new host. It may, however, use the names of any singleton Actors (Actors with at most one instance per host) on the new host. The second restriction on movement is that the Actor which is to be migrated must be known by (i.e. an acquaintance of) no more than one Actor. As Sackman points out these restrictions are to aid simplicity and avoid any 'under-the-bonnet magic'.

4.4.3 Syntax

Listing 20 A (confusing) GLINT program [68].

```
1  define Counter(Number value)
2  become new Counter<value'>
3  where
4    value' = value.plus<One>
```

Listing 20 shows the definition of a simple (count up) counter in GLINT. This example shows that a definition of a simple Actor can be confusing in GLINT. Line 1 begins the Actor definition and states that a counter Actor is parameterised by a 'value' of type 'Number'. At this point it should be noted that whilst syntactically GLINT supports typing of variables no type checker is provided. Lines 2 though 4 specify the replacement behaviour for the counter upon receipt of an anonymous message. Considering a counter instance, c, we can send an anonymous message using the following statement: `c' = c<>`. Listing 21 gives a definition of a 'wrapping counter'.

Listing 21 A more complicated GLINT program.

```
1  define WrappingCounter(Number value, Number max)
2    inc()
3    inc value.lessThan<max>
4      become new WrappingCounter<value'>
5        where
6          value' = value.plus<One>
7    inc
8      become new WrappingCounter<Zero>
```

It is parameterised by both an initial value (`value`) and a maximum value (`max`)[3]. When the counter's value reaches the value of `max` the counter 'wraps' back to zero.

The code for incrementing the counter has been extracted into its own method: `inc`. Whilst this is a method it is not invoked via 'usual' local calling procedures rather it is executed on receipt of an `inc` message. This forgoes any programmatic inspection of messages: the method type and parameter are extracted from the message automatically.

[3]We assume as a precondition that $value < max$.

Specifying behaviour in terms of methods allows Actors which are to interact with an Actor to simply make 'method calls' rather than possess knowledge about the format of messages the target Actor is expecting to receive. Most programmers are more familiar with method calls than message passing and an Actor's methods form a solid and tangible interface. Method calling in GLINT is synchronous, so-called call and return, although asynchrony can be achieved by calling a 'void' method on an Actor as these return immediately. This is a questionable design decision. Synchronous method invocation can lead to deadlock. Listing 22 shows a seemingly innocuous encoding of a pair of Actors who 'ping-pong' each other which will immediately reach a deadlocked state.

Listing 22 Deadlock in GLINT.

```
define A(Actor b)
 b.ping<>
 pong()
 pong
   b.ping<>

define B(Actor a)
  ping()
  ping
    a.pong<>
```

Returning to the counter (listing 21), branching is encoded using guards. Guarded expressions are also found in HASKELL (example shown in listing 23) and provide a clear and readable way to distinguish alternative behaviours. The GLINT example has two guards: one for 'normal' incrementation behaviour and one for the 'wrapping' behaviour.

Listing 23 Guarded expressions in HASKELL.

```
fact n
  | n == 0    = 1
  | otherwise = n * fact (n-1)
```

4.4.4 Evaluation

GLINT was designed as a prototype language to demonstrate pertinent features rather than as a complete language so we can forgive many of its syntactic blemishes. Using message passing to implement method calls unburdens the programmer from having to explicitly deal with the format of messages. Synchronous invocation using messages is a useful feature for an Actor language although the benefits must be weighed against the potential for deadlock.

4.5 Scala

SCALA [62] is an Object Orientated, component orientated, functional programming language capable of running on both the JVM and the .Net platform. The language has some suitably grand aims, notably it aims to fuse object orientated and functional programming. Many commentators have noted that JAVA considers functions only as a part of an object and not as objects

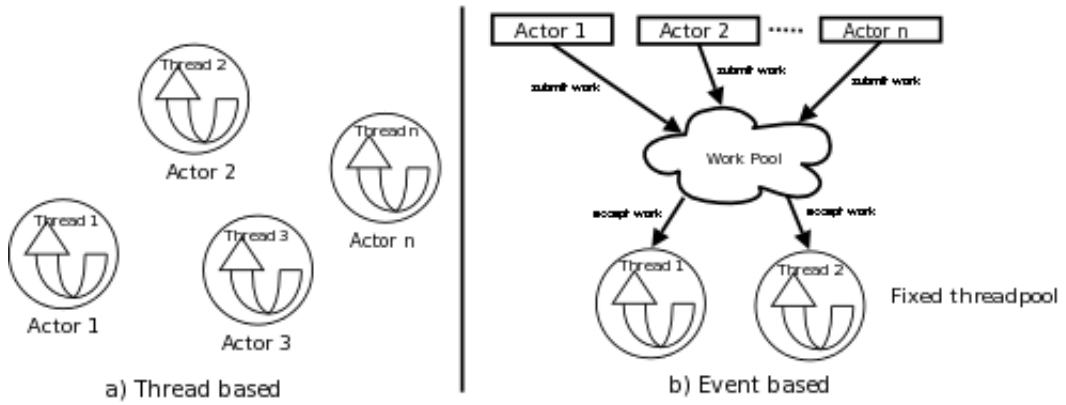

Figure 4.3: Event-based and thread-based Actors.

in their own right. Steve Yegge makes the point better than I can in his infamous 'Execution in the Kingdom of Nouns' [81]. SCALA attempts to redress the balance by introducing constructs more typically found in functional languages, including first-class functions and partial functions, to a JAVA-like system. This leaves the language, in my opinion, as an unhappy mix of two paradigms.

In this section we shall not concern ourselves further with the SCALA language, rather we consider the event-based Actor extensions provided by Philipp Haller in [37].

4.5.1 Event based Actors

Implementers of Actor languages face an important design decision early in development. This decision is whether to assign one thread per Actor or to use an event based model where Actors submit work to a pool of threads which perform the work. The alternate models are shown graphically in figure 4.3. In this section 'thread' should be read as 'a thread as presented by the underlying machine/language'. For example in JAVA 'thread' pertains to a JVM `Thread` (which is usually an Operating System thread). We note that it is the task of an event based language's runtime to assign threads to Actors dynamically - such a system is a user-space scheduler. Schedulers are non-trivial to implement. The advantage of an event-based model is that there is a negligible Actor switching overhead (the time taken the transfer control of the CPU from one Actor to another) and Actor creation time is negligible since it only involves initialising state rather than creating a new thread.

4.5.2 Unifying threads and events

In [39] Haller and Odersky show how event-based Actors and thread-based Actors can co-exist. To achieve this unification SCALA Actors may use the following primitives:

- `send` - Non blocking send. This semantic is found in ERLANG and most other Actor languages - I have not encountered an Actor language with a synchronous `send` primitive[4].

- `receive` - Wait for a message by blocking the current thread. Execution continues from the instruction which follows the receive statement.

- `react` - Programmers specify the closure which is to be executed when a suitable (matching some criteria) message is received. Actor execution resumes by executing the closure. The Actor's future behaviour is completely defined by the closure. All contextual information, including the stack, is abandoned.

`react` allows SCALA Actors to (in effect) 'give up' their thread. When waiting for a message an (event based) Actor exists only as a closure stored in memory. Importantly when an Actor is in this state it has no thread associated with it. When an Actor sends a message it looks up the target Actor in memory and if the target Actor is waiting on a message it selects a thread from a thread pool and 'revives' the recipient. Hence the designation 'event based Actors'. This model is advantageous because we need fewer (JVM/OS) threads. Similarly since a thread can complete the execution of more than one Actor's behaviour during its CPU timeslice the inter-Actor switching overhead is reduced.

Event-based Actors in SCALA [38] are implemented in-language which makes scheduling more restrictive since the scheduler can only access information available from *within* the language. The scheduling techniques used are not presented in full, however from the information presented the method seems inelegant. Notably exceptions are used to unwind the call-stack and the scheduling is not preemptive. Actors retain their thread until they use the 'react' primitive or they reach the end of their body. Actors whose behaviour follows a `while(true) work()` pattern are Actors which present no opportunities for SCALA to reschedule the underlying thread. Pure calculation tasks often follow this pattern, and as such will 'hog' a JVM thread. SCALA Actors are free to make blocking calls. Calls which block an Actor for long period of time result in the same unresponsive behaviour.

The potential for such unresponsiveness precludes a fixed threadpool. To demonstrate this we consider thread pool of size five (i.e. the pool has five threads at its disposal). If five Actors become unresponsive (either by becoming work-bound or blocking) a 'system induced deadlock' arises. In this situation Actors are willing to run but there are no threads left in the pool to run them. A further complication is that the 'scheduling' is done in-language and there is no way to reliably detect when another thread is blocked. The SALSA scheduler uses timestamps to approximate when a thread was last active, threads whose timestamps has not been updated *recently* are assumed to be blocked. If all threads are considered unresponsive then a new (JVM) thread is spawned.

[4]Many do have synchronous invocation but this is a higher-level abstraction which is usually implemented on top of an asynchronous send primitive.

4.6 Designing a language

> "There is no programming language, no matter how structured, that will prevent programmers from making bad programs." [28].
>
> *Lawrence Flon*

Using and evaluating existing and established Actor-based languages highlights useful, confusing and poorer features of these languages. This research lead us to a set of guiding principles and design goals for STAGE. A summary of the design goals is presented below.

- STAGE should avoid mixing paradigms. SALSA exhibits this behaviour: continuation passing requires a different mindset to that of JAVA which co-exist in SALSA. STAGE should ensure that its abstractions, operators, and style are consistent.
- Of the languages presented, ERLANG is the language most suitable for fast development of concurrent systems. This is unsurprising as it is a syntactic cousin of PROLOG, renowned for encouraging fast development. ERLANG's lack of support for arbitrary code migration and process naming and discovery means that more general distributed systems require additional 'scaffolding' or components. STAGE should have a lightweight syntax and support for high level distribution constructs.
- More theoretically orientated Actor languages such as GLINT provide no, or limited support for Actor interaction with the devices of their local host. To be usable STAGE must provide such interactions in a manner which is consistent with the Actor model. Device Actors (similar to native methods of JAVA) can be used to span the conceptual void between the Actor world and the underlying Operating System.
- STAGE must be able to exploit the potential parallelism in multiple core architectures. We note that early ERLANG implementations could not achieve such speed ups. We must also ensure that STAGE is not prohibitively slower than other languages that provide similar features.
- STAGE should avoid boilerplate code. Boilerplate code is code which appears throughout a program with little change. Boilerplate often occurs in situations involving I/O although not exclusively. Listing 24 shows the boilerplate required in JAVA to read a file of integers and its PYTHON equivalent. The undesirability of such boilerplate is well known: 'Boilerplate code is tiresome to write, and easy to get wrong. More-over, it is vulnerable to change.'[48]. STAGE should avoid encouraging boilerplate where possible.
- Some frameworks for distribution require participating clients to generate code based on an abstract definition of remote interfaces. Many web services are defined using the Web Services Definition Language (WSDL). A common way to interact with such services is to generate local 'binding objects' which represent the remote resource. These binding objects are usually generated at compile time. A popular tool for generating bindings for JAVA is the JAXB[5] compiler. The language should avoid forcing programmers to explicitly generate code to interact with remote Actors.
- It is reasonable that each host be running an interpreter for STAGE but there should be few other components required for a STAGE program to execute successfully.

[5] See http://java.sun.com/webservices/jaxb/.

Listing 24 Reading a file of numbers in JAVA and PYTHON.

a) In JAVA

```
static List<Integer> readFile(String fileName) throws NumberFormatException, IOException {
    List<Integer> numbers = new LinkedList<Integer>();
    BufferedReader in = new BufferedReader(new FileReader(fileName));
    String line;
    while((line = in.readLine())!= null) {
      numbers.add(Integer.parseInt(line));
    }
    return number;
  }
```

b) In PYTHON.

```
  def read_file(file_name):
    return [int(line) for line in open(file_name)]
```

Chapter 5

The Stage language

This chapter outlines both the STAGE language and its underlying implementation. It is hoped that this section provides sufficient information for a potential STAGE programmer to become familiar with the syntax and style of the language. I will discuss the design decisions, strengths and limitations of the language and provide justification where appropriate. STAGE is a language embedded in PYTHON and familiarity with PYTHON will aid comprehension, however such background knowledge is not a prerequisite. STAGE's features include:

- Synchronous and asynchronous communication.
- Unrestricted migration.
- Distribution and inter-host communication.
- Rendezvous and Actor discovery.
- Load balancing and host monitoring.
- Real-world interaction including graphical elements.
- Event handling and non-blocking Actors.
- A complete set of example applications.

STAGE runtime systems have two distinct components. Firstly there is the STAGE execution environment, known as a theatre. By running a theatre hosts are able to harbour Actors. A theatre allows a host to accept migrating Actors and to migrate Actors to the network as shown in figure 5.1. A theatre provides support to Actors executing within it. This internal support facilitates Actor behaviour including creating Actors, sending messages and inter-host migration. The execution environment (shown in figure 5.2) includes meta-logic which subverts PYTHON calling conventions and instantiation routines to support the Actor model. Networks containing STAGE theatres must also contain locator components. These track Actors' movements through the network and support rendezvous. They are described fully in section 5.4.2.

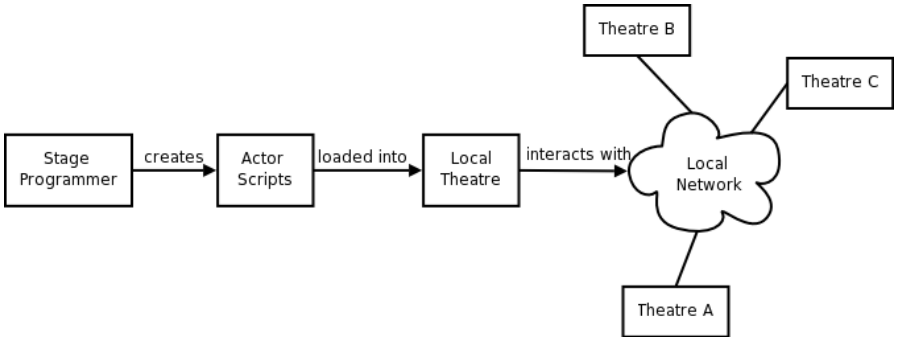

Figure 5.1: A typical STAGE workflow.

5.1 Implementation

When designing STAGE our intention was for the language to be based on PYTHON using rewriting to add Actor language constructs. The motivation for this approach came from SALSA, which is an Actor language based on JAVA, whereby SALSA Actor definitions are rewritten into standard JAVA classes. After investigating PYTHON further it soon became clear that due to the dynamic and flexible nature of PYTHON with its meta programming capabilities such rewriting would not be required. This is the same approach that SCALA event-based Actors use - Actors are implemented in-language with no rewriting.

How does PYTHON support meta-programming? Name binding concerns the binding of names to declarations. Applied to objects, a name binding strategy defines a mapping between a method call to an object and the concrete implementation of the method. JAVA binds early - if a call is made to an object then the method's implementation must be defined by the class or inherited. This has the advantage that it allows compile-time checking, at the cost of flexibility. The flexibility of PYTHON stems from its late binding. In general to call a method a caller requests the method *by name* from the object. The object responds with a `callable` object (which represents the method) bound to the specific object instance or raises an exception if it cannot fulfil the request. The caller proceeds by executing the 'call' method of the `callable` it received. This allows an object to respond to an arbitrarily named method call. We utilise this behaviour to insert dynamic proxies throughout the system which modify PYTHON's calling conventions. We use a similar technique to implement naming - Actors are contacted by name and not by reference. We also use this technique to prevent Actors from sharing state by preventing direct method calls - all communication is via messaging and copying. PYTHON provides the ability modify object creation. We use this feature to enforce Actor instantiation rules. Migration is achieved by a complex combination of runtime reflection and dynamic class loading. The theatre runtime itself, the communication libraries, migration engine and location services are all implemented in PYTHON.

There are two interrelated advantages to this approach. Firstly the language maintains most of PYTHON's constructs and features. The second is that existing PYTHON developers will immediately comprehend the syntax. The benefits are discussed further in chapter 7 but for now I will focus on those features of PYTHON which are supported and those which are not.

All of PYTHON's key constructs are supported. It is far simpler to list those features of PYTHON

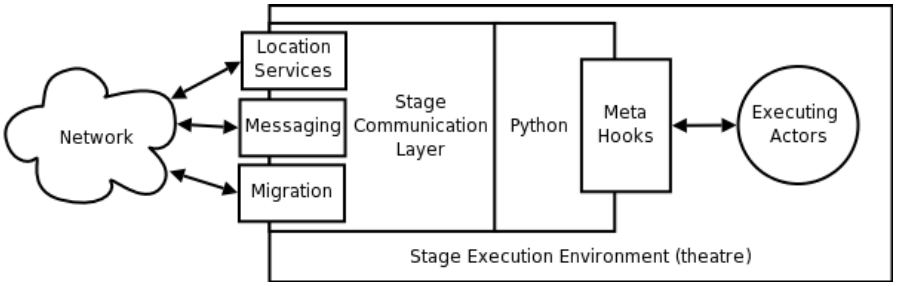

Figure 5.2: The STAGE execution environment.

which are not supported rather than give a complete list of PYTHON's features. Removed features fall into three categories: threading, first class functions and external interaction. PYTHON's threading libraries are redundant since the Actor language constructs provide the same operations. Use of the `threading` library is not supported since it can interfere with the runtime environment's threading model. For the same reasons generator expressions are prohibited. Methods in STAGE are *almost* first class. In STAGE methods can be passed and stored in the same way their PYTHON counterparts can. The only restriction is that to pass or store a method the method must be part of the external interface of an Actor - this precludes defining anonymous functions in inner scopes. Finally all external interaction must be via Driver Actors which moderate Actors' communication with the outside world. This stops Actors engaging in operations which may leave them in an unresponsive state (e.g. blocking I/O) or may leave a local resource (e.g. a file) in an inconsistent state. Driver Actors are discussed further in section 5.4.3.

5.2 Actors in Stage

5.2.1 Defining an Actor

Actor definitions take the same form as class definitions in PYTHON. Defining a method on a Actor represents an Actor's capability to accept, process and possibly respond to a message of a specific type. Listing 25 gives a definition, instantiation and use of a `Dog` Actor which exposes two capabilities: 'walk' and 'feed'.

Lines 1 through 13 specify the behaviour `Dog` Actors must follow. This is termed the 'script' of a `Dog` Actor - all *correct* dogs will behave in this way. Line 1 declares that `Dog` Actors are mobile Actors as defined in section 5.2.2. For now consider this declaration as giving `Dog` Actors the ability to migrate between hosts.

Line 3 specifies the special 'birth' method. This is executed when an Actor instance is instantiated. It is guaranteed to be the first method executed in an Actor's life cycle.

Lines 7 through 13 specify the external interface of `Dog` Actors and the behaviour `Dog` instances should exhibit upon receipt of `walk` and `feed` messages.

Listing 25 An Actor definition in STAGE.

```
1  class Dog(MobileActor):
2
3    def birth(self):
4      self.in_motion = False
5      self.hungry = False
6
7    def walk(self):
8      self.in_motion = True
9      return 'woof woof'
10
11   def feed(self, bowl, food):
12     self.hungry = False
13     bowl.remove(food)
14
15 rover = Dog()
16 rover.feed(bowl, food)
17 print rover.walk()
```

Readers unfamiliar with PYTHON should note that `self` (which refers to an Actor instance) is explicitly passed as the leftmost parameter. Those familiar with JAVA and C++ may find this unsettling since JAVA and C++ pass a `this` pointer implicitly. Furthermore since PYTHON does not permit variable declarations, access to instance variables and methods must be made explicitly through the `self` variable. Lines 4 and 5 initialise instance variables `in_motion` and `hungry`, omission of the prefix `self` would result in `in_motion` and `hungry` being considered method-local variables rather than instance variables. This is a much debated issue within the PYTHON community and is unlikely to change.

Listing 26 shows the same code presented in ERLANG. This highlights how STAGE's Actor definitions promote structure and readability. A comparison of the STAGE and ERLANG implementations follow.

- In ERLANG the atoms `walk` and `feed` are used to distinguish messages however this is a convention used for this example and not enforced syntactically. As a result the external interface of the Actor is not immediately obvious.
- The walk action 'returns' a result. This is achieved in ERLANG by explicitly passing the caller's PID (process identifier) such that the Actor can send a 'result' message back to the caller. The caller must perform a receive operation to determine this result. If the callee is expecting results from multiple Actors it may be necessary to include request identifiers to distinguish results. This couples the caller and the callee to a specific interaction protocol which is not enforced syntactically.
- We must explicitly pass all the state of the process each time we modify either `hungry` or `in_motion` even though we may not modify both.
- Passing a message which the process cannot handle, i.e. a message which matches none of the message patterns results in the message sitting in the process' queue indefinitely. STAGE presents an error to caller and the message is removed from the queue.

I do not wish readers to leave this section with the belief that I am claiming ERLANG is in anyway inferior STAGE. There are situations, especially those where message traffic must be minimised, where ERLANG's `send/receive` interaction style is more suitable.

Listing 26 A Dog Actor in ERLANG.

```
dog(Motion, Hungry) ->
  receive
    {walk, Caller} ->
       Caller ! "woof woof",
       dog(Motion, true);
    {feed, Bowl, Food} ->
       Bowl ! Food,
       dog(Motion, false)
  end.

run() ->
  Pid = spawn(animals, dog, [False, False]),
  Pid ! {feed, bowl, food},
  Pid ! {walk, self()},
  receive ->
    Result -> io:write(Result)
  end.
```

5.2.2 Actor ontology

"All Actors are equal, but some Actors are more equal than others."

With apologies to George Orwell

A mobile Actor is the most general variety of Actor. Such Actors may move freely from theatre to theatre without restriction. If an Actor wishes to send a mobile Actor a message then it must know the unique name of the mobile Actor. An Actor learns the name of a mobile Actor by either being the parent of the Actor or by learning the name through Actor-interaction. A `LocalActor` has all the properties of a `MobileActor` except that it cannot move from its originating host. `LocalActor`s represent Actors that have no need to move or require contextual information from their originating host - they would cease to function correctly on a new host. Where possible Actors should be designated `LocalActor`s since these have the most efficient implementation.

Singleton Actors embellish the pure Actor model by allowing interaction with Actors though a known name. Such Actors have proved invaluable in ensuring STAGE programs are conceptually concise. They are motivated by two use-cases. Firstly where a real-world abstraction has exactly one instance such as a console input Actor or a screen (display) Actor. Secondly when it is necessary to provide a named access point to a service such as a file system or higher-level communication service (e.g. a publish/subscribe access point). STAGE provides two scopes: Local and Network. Singleton Actors with local scope (`LocalSingletonActor`s) provide local access points. For example interacting with a `Clock` Actor will yield different results on different hosts since Actors will be communicating with host-specific local instances. A `NetworkSingletonActor` is an Actor which represents a unique name within a network. These are most commonly used to facilitate Actor rendezvous.

`NewtorkSingletonActor`s should be used sparingly and with care. By their very definition only one instance of an `NetworkSingletonActor` type exists within a network. If not used judiciously such Actors can become a bottleneck. If a `NetworkSingletonActor` is subject to a high request load then the performance of the whole Actor network will suffer. Furthermore they represent a single point of failure in a network. For the reasons outlined

Listing 27 A LocalSingletonActor in STAGE.

```
class FileSystem(LocalSingletonActor):

  def get_file(self, filename):
    """Returns a file Actor which represents the file"""
    (implementation omitted)


  def file_exists(self, filename)
    """Returns True iff the file exists on the local filesystem"""
    (implementation omitted)
```

Listing 28 A NetworkSingletonActor in STAGE.

```
class Games(NetworkSingletonActor):

  def add_game(self, gameserver):
    """Register a new game server"""
     (implementation omitted)

  def a_game(self):
    """Returns a game server"""
    (implementation omitted)
```

`NetworkSingletonActor`s should only be used to unite Actors with (non-singleton) Actors that are able to satisfy further requests - the `NetworkSingletonActor` should not perform operations itself.

5.3 Language features

5.3.1 Asynchrony

Calls made to an object instance in most Object Orientated languages are synchronous since the calling object's thread enters the callee and executes the body of the callee's method. For this reason the calling object cannot continue its execution until the result is determined. In some sense the 'default' interaction style is synchronous: designers must requisition bespoke objects and threads to provide asynchrony on top of this synchronous base.

STAGE takes the opposite view. Every Actor-Actor interaction within STAGE is, by default, asynchronous. Execution of the calling Actor continues whilst the callee processes the message and generates the result. Of course, the method call abstraction is somewhat less useful without the ability to gather the result. One of the greatest determinants of Actor-based languages is the method by which results are retrieved and the synchronous-asynchronous divide is not a dichotomy rather a continuous axis. GLINT provides only synchronous invocation but allows a callee to 'return early' in certain circumstances (thus releasing the caller) whereas SALSA's primary behaviour is asynchrony via token passing continuations.

The motivation for making asynchrony the default behaviour is to encourage independent computation and parallelism which leads to performance gains. STAGE provides both asynchronous and synchronous result capture which are discussed in the following sections.

5.3.2 Waiting for a result

The `sync` keyword causes the calling Actor to block until the callee returns a result. This turns STAGE's asynchronous call semantics into a more familiar call-wait-return semantic, mimicking PYTHON's call and return. By its very nature `sync` imposes restrictions on parallelism and should be used conservatively and with care - a blocked Actor cannot service its message queue. It should only be used when the result is *actually* required. The keyword can be used at any point - not just at the call-site. To make this clearer we consider a (trivially simple) 'bank' Actor whose interface consists of one method: `total()` which returns the total amount of money currently in the bank. To place the result of calling `total()` into a local variable a client Actor can use call-site synchronisation as shown in listing 29

Listing 29 Call-site synchronisation.

```
1  money = sync (bank.get_money())
2  ... code continues ...
3  print money
```

This is call-site synchronisation, and should be avoided where possible since there is no scope for parallelism in such interactions. Listing 30 demonstrates that in STAGE synchronisation can be applied right up to the point at which a result is required.

Listing 30 Late synchronisation in STAGE.

```
1  money = bank.get_money()
2  ... code continues ...
3  print sync (money)
```

In listing 30 the actual computed value of `money` is not required until line 3 (where it is printed). Thus there is no need to wait for the result (using `sync`) until the print statement. Notice that after line 1 and until line 3 the variable `money` contains a partial or 'marker' result. This construct draws inspiration from 'futures' (section 2.2.1) which are a receipt for work not yet completed. Figure 31 is a partial solution to the work distribution example (section 3.3.3) which demonstrates the `sync` keyword being used to wait for the completion of multiple invocations. This usage mirrors join continuations in SALSA (section 4.3.3).

Listing 31 Waiting for multiple results in STAGE.

```
1  markers = [ Marker(exam) for exam in exams ]
2  results = sync ([ maker.mark() for marker in markers])
3  print results
```

The need for manual synchronisation can soon slow development as it is both a typographical, conceptual and cognitive hindrance. A programmer must determine the 'latest' point at which synchronisation should occur to maximise parallelism. Similarly when modifying code programmers must be careful to modify the synchronisation sites accordingly. A solution that emerged during development is named 'automatic lazy synchronisation' and is presented in section 5.3.5.

5.3.3 Polling

When reading this section consider a HTTP request for a heavyweight operation. For a real-world scenario consider a transport planning website whereby a user enters an origin and destination and is returned a sensible route from the origin to the destination. The planning will take some time, and user is presented a 'loading' page whilst the plan is computed. To implement this such sites often poll (those with a penchant for web 2.0 can consider AJAX polling if they wish) the server and ask 'is the result ready yet?'.

Such interactions occur frequently in applications where a request-response interaction is employed. This interaction benefits scalability since a persistent two-way connection is not required. To support such interactions in STAGE we provide the `ready` keyword. The `ready` keyword takes a partially computed result and determines if the result is ready yet. A typical interaction is shown in listing 32.

Listing 32 Testing the status of a result.

```
1  money = bank.get_money()
2  ... code continues ...
3  if ready (money):
4    print "There is  %d in the bank." % money
5  else:
6    print "I don't know how much money is in the bank yet."
```

5.3.4 Event handlers

In STAGE a handler is a method which is 'installed' and automatically executed upon receipt of a suitable message. It is inspired by SALSA's token passing continuations and Haller's event based Actors for SCALA. STAGE's implementation is, in my opinion, more intuitive although arguably less general. A handler is installed using the `handle` primitive. One must specify a message handle (the partial result) and a method to be executed upon receipt of a message. Listing 33 shows how a client can interact with the bank (of listing 32) using a handler.

Listing 33 Installing a handler in STAGE.

```
1  def query_bank(self):
2    money = bank.get_money()
3    handler(money, self.query_complete)
4
5  def query_complete(self, amount):
6    print "There is  %d in the bank" % amount
```

A handler provides greater scope parallelism since it is event based - Actors do not block whilst they wait for a result. This leads to responsive Actors and should be preferred. However message handlers have a distinct disadvantage - loss of context. A `sync` statement receives the result and the result can be processed with the current stack intact. The handler specifies which code should be run on receipt of a message - it does not restore the calling context. The concept and concurrency gains of a message handling style are borrowed from event based Actor implementations [39].

5.3.5 Lazy synchronisation

Explicit synchronisation using the `sync` keyword increases the conceptual overhead, reduces clarity and is unwieldy for chained Actor calls. Chained Actor calls are those whereby an initial call on an Actor is performed and a further call is made immediately on the result to form a chain of arbitrary length. Such chains are often necessary in code where multiple layers of indirection have been employed. The use of fluent interfaces [30] in STAGE generate such chains. Lazy synchronisation is STAGE's ability to defer waiting for a result until the result *actually* is required. Listing 34 shows the same code both without lazy synchronisation (line 1) and with (line 2).

Listing 34 Lazy synchronisation simplifies chaining Actor calls.

```
1 print sync(sync(sync(operator.trains()).fastest()).has_dinner_service())
2 print operator.trains().fastest().has_dinner_service()
```

An Actor may be required to gather information from its acquaintances and then forward either the unmodified information or the result of processing the information to another Actor. To demonstrate the usefulness of lazy synchronisation in this context consider an Actor whose task is to retrieve and sort a list of exam marks (from the acquaintance `marks`) and the teacher (from the acquaintance `teachers`). Without implicit synchronisation the most likely solution would be to retrieve the marks and the teacher using join synchronisation. However this restricts potential parallelism as the Actor must wait for both the teacher and the marks before it can begin sorting the marks. If the `teachers` Actor is on a host with high load or a poor (low bandwidth or high loss) channel then the sorting is delayed until the `teachers` Actor's result message is received. With lazy synchronisation (listing 35) the Actor only waits for the marks when they are required, which will be when the sorting routine attempts to access an element. The important difference is that the Actor need not wait for the teacher until line 4, which is the first time the value is required. Here the use of implicit lazy synchronisation has allowed the sorting to take place before the teacher is determined. Performance will increase as the sorting and waiting for the `teachers` Actor to return the relevant teacher effectively happen in parallel.

Listing 35 Implicit synchronisation improves performance.

```
1 marks = marks.get(classname)
2 teacher = teachers.get(classname)
3 sorted_marks = sort(marks)
4 headteacher.send(teacher, sorted_marks)
```

The disadvantage of such a scheme is the loss of explicit `sync` statements. Without lazy synchronisation Actors can only block where a `sync` statement[1] is present. When a deadlock is encountered it the source can be immediately attributed to a rogue or misused `sync` statement. Furthermore modelling STAGE programs (should one wish to do so) is simpler without lazy synchronisation for similar reasons. The decision to include such a feature reduces to a decision as to whether the extra conciseness and clarity afforded outweighs the dangers of implicit

[1] This restriction is lifted for local Device Actors.

synchronisation. As the author of the language and its sole developer I have experienced and experimented with both styles. The feature is powerful enough to eliminate the need for explicit synchronisation and the associated extra tokens which accompany it. I will concede that I have experienced deadlock as a result of this mechanism but I was able to quickly resolve it. As a result lazy synchronisation remains an integral and useful part of the STAGE language.

5.3.6 Messages and order

To discuss STAGE's message order policy we consider a `customer` Actor whose *raison d'etre* is to add items to a `shopping_cart` Actor before finally requesting the total cost of shopping in the cart. An example interaction is shown in listing 36. Lines 1 through 3 result in three 'add'

Listing 36 Shopping with Actors in STAGE.

```
1  cart.add("beans")
2  cart.add("bread")
3  cart.add("milk")
4  total = cart.total()
```

messages being sent to the `cart` Actor. In many languages the order in which these messages arrive at the recipient is not guaranteed. In STAGE the order (in the absence of failures) is guaranteed to be the order in which they are sent. In the example this means that the final 'total' message on line 4 **cannot** be received before all the 'add' messages have been received.

More generally if an Actor, A sends messages $M_a = a_1, a_2, ...a_n$ to Actor B then B receives M_a. If a third Actor, C sends messages $M_c = c_1, c_2...c_n$ then B receives an arbitrary and non-deterministic interleaving of M_a and M_c which preserves the independent orderings of M_a and M_c. These orderings are enforced by the STAGE environment. Preserving order suffers a small performance penalty but I have found that this guarantee has avoided the pitfalls prevalent in languages which, for whatever reason, do not make this guarantee.

5.3.7 Methods

In STAGE each Actor exposes a number of methods which other Actors may invoke. During development of STAGE one interaction pattern became prevalent: the callback. In this pattern Actors send their name to other Actors and after some time other Actors 'callback' the Actor when either a result is ready or an event has occurred. Figure 37 shows a TextEditor Actor who requests a callback when a file changes. The problem with this approach is that the TextEditor sends its name to the FileSystem and eventually the FileSystem invokes a known method (`file_changed`) on the target (the TextBox). This represents high coupling since the target **must** have this method defined and the FileSystem **must** know which method to call. If the TextEditor has asked for callbacks from multiple sources then its interface become brittle. Similarly if a new version of the FileSystem Actor is released which calls `change` rather than `file_changed` then all the clients of the FileSystem (including the TextEditor) are broken.

The solution proposed, and implemented in STAGE, is to promote the status of methods. Methods in STAGE are not strictly higher-order (since they cannot be generated on the fly) however

Listing 37 Requesting a callback in STAGE.

```
class TextEditor(LocalActor):

  def brith(self, filesystem):
    filesystem.notifyWhenChanged(self, 'foo.txt')

  def file_changed():
    ... handle change here ...
```

they share a number of useful attributes. Methods can be passed freely between Actors, stored in instance and local variables and are preserved under migration. Actors can even pass methods that form part of other Actors. Listing 38 shows how a WebBrowser Actor tells a DNS Actor to send the IP address to a PageFetcher Actor. Callbacks can be generalised into a publish/sub-

Listing 38 Passing functions to Actors.

```
class WebBrowser(MobileActor):

  def request(self, url):
    ...
    dns_server.lookup(page_fetcher.fetch)
    ...
```

scribe mechanism where Actors specify which events they are interested in. Actors subscribe to an event type by sending a method to the publish/subscribe server. If the relevant event occurs the method is executed. This pattern occurs frequently in STAGE programs and as such a skeletal publish/subscribe mechanism is included in the core library.

5.3.8 Migration

STAGE, like its predecessor, GLINT, embellishes the pure Actor model by including both distribution and migration. Distribution allows Actors on remote theatres to interact with any other Actor, making no distinction between local and remote Actor-Actor interaction. Migration allows Actors to move freely between hosts. STAGE supports migration through the `migrate_to` primitive. The body of an Actor executes until it reaches a `migrate_to` statement. Upon executing this statement the state of the Actor is frozen and migration is attempted. At this point an Actor's execution may proceed in one of two ways. Firstly the migration may succeed. In this scenario the Actor resumes its execution in the new theatre. Since STAGE provides weak mobility execution continues from code placed in the `arrived` method. The original implementation allowed the specification of an arbitrary function which the Actor should execute upon arrival. In practice this results in confusing code, since it is not immediately obvious (by looking at the code) which method is executed upon arrival at a host. For this reason the `arrived` method is the only method that resumes an Actor's execution upon arrival at a theatre. If a migration attempt is unsuccessful the `migrate_to` keyword returns an error code and the Actor's execution continues on the original host.

Cardelli and Gordon [21] consider both objective and subjective movement. Their discussion is in the context of the ambient calculus but the distinction can be applied to the Actor model. A subjective move (or migration) is a movement initiated by the Actor which is to move. An ob-

jective move is a movement initiated by another Actor, in Cardelli's words 'I move you'. STAGE supports subjective movement since every Actor (more correctly every mobile Actor) can move itself using the `migrate_to` keyword. Objective movement is supported if the Actor which is to be migrated allows itself to be moved. Put simply since migration can only initiated using the `migrate_to` keyword Actors can expose this ability by placing it in a publicly accessible method. This is analogous to an object (in an Object Orientated language) exposing a capability by placing an operation in a `public` method.

Friends

> "Ignoring the difference between the performance of local and remote invocations can lead to designs whose implementations are virtually assured of having performance problems" [76]

Whilst STAGE makes no distinction between local and remote Actor interaction, a message which spans hosts (a remote message) is around four to five orders of magnitude slower than a local interaction [76]. For this reason it is desirable that groups of Actors which engage in high-volume communication be located in the same theatre. We refer to such groups as Actor cliques. To ensure that such Actors remain together an Actor may specify its friends. The STAGE execution environment attempts to keep cliques on the same theatre where possible. The current implementation is naive - if an Actor specifies that another Actor is its friend the Actors are moved together. Errors are raised if an attempt is made to move an immobile Actor. Manual specification of 'friends' is error prone since forgetting to include a friend means the omitted friend is not migrated. This can result in accidental high-volume network communication. For this reason automatic identification of 'friends' should be investigated and included in future releases.

5.3.9 Apoptotic Actors

An Actor's life-cycle is usually follows three distinct phases. (1) birth and initialisation, (2) useful work and interaction and finally (3) death. Most languages deal with phases (1) and (2) but ignore (3) for reasons outlined shortly. If STAGE is to be a usable language, rather than a demonstration language, we must accept that any program which continually generates Actors will eventually run out of resources if old Actors are not removed. Removing Actors which no longer fulfil a role ('garbage Actors') is analogous to garbage collection in an Object Orientated language. JAVA and PYTHON programmers are not required explicitly harvest unused objects, they are automatically removed by a process called garbage collection. Other languages such as C and C++ require programmers to explicitly free memory used by defunct objects[2].

STAGE aims, where possible, to be a usable and concise as possible and as such should provide garbage collection for Actors. However garbage collection in a network of mobile code faces the following challenges [78]:

[2]Garbage collectors do however exist for these languages.

- Distribution - No one node has complete information about the network. Garbage collection must proceed through inter-host communication, collaboration and agreement.
- Synchronisation - Lack of a global clock or synchronous heartbeat, decisions are based on asynchronous messaging. Similarly stopping execution of all hosts to perform a collection is prohibitively invasive.
- Concurrency - Actors execute independently and can dynamically change their collaborators.
- Failure - Hosts may fail during collection or may give incorrect information.
- Migration - Actors in the process of migration may hold references to an otherwise unreferenced Actor.
- Ambiguous garbage - When is an Actor garbage? When is it doing useful work? These questions cannot be answered trivially.

Listing 39 The act-die behaviour of a MaintenanceActor.

```
1 ...
2 def clean_system(self):
3   filesystem.delete("/tmp/*")
4   buffers.flush()
5   die()
6 ...
```

Building such a collection system under these conditions would be a project in itself. STAGE sacrifices some simplicity and supports only explicit Actor suicide. The `die` primitive causes an Actor to terminate. Listing 39 gives a (partial) specification of a `MaintenanceActor` whose task is to clean exactly one host. Once its role is fulfilled, i.e. it has cleaned the system, it can die as it is of no further use. Not all Actor death scenarios are as simple as this act-die cycle and Actors may need confirmation from their clients that they are able to die. In practice this mechanism is sufficient to prevent a build-up of unwanted Actors but I have encountered situations where many small Actors were created and never died resulting in theatre failure.

Load balancing

The STAGE execution environment, by default, does no load balancing. STAGE provides a useful set of libraries which can be used to implement load balancing. Since there are a diverse range of load balancing techniques and strategies I felt it was more appropriate to provide load balancing as libraries which can be extended and modified rather than in the core language.

STAGE provides a `ProbingLoadBalancer` as an example implementation. It consists of the following components:

- An interface Actor which controls registration of worker Actors (those Actors which have expressed a willingness to be load balanced).

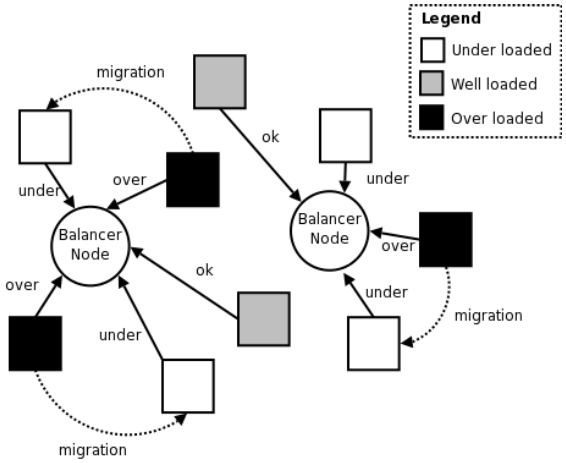

Figure 5.3: STAGE's load balancing strategy.

- Strategy (or 'decision') Actors which determine if and when Actors should be moved. These have access to estimates of Actors' current locations and information gathered from theatre probes (below). These Actors can be modified, swapped or removed to optimise the strategy.

- Monitoring probes which report back a measure of the load on a host.

The implementation provided uses a simple decision procedure based solely on theatre load as shown in figure 5.3. We consider only CPU load and each host's load is characterised as either (1) underloaded, (2) well loaded, (3) overloaded. Conversion between the reported load, a continuous measure, and the load status (one of the three characterisations) is based on reasonable assumptions and empirical investigation. The load balancer continually moves Actors from 'overloaded' hosts to 'underloaded' hosts. The load balancer can be considered short-sighted as it does not attempt to group communicating Actors rather it attempts to distribute the load evenly. Even with these limitations this scheme is successful and the results are presented in section 6.2.5.

More complicated techniques are possible, and implementers are free to add new strategies. An interesting strategy is to incorporate message volume information [25]. However more complicated strategies are usually at the expense of speed (the time taken to make a decision) and generality (which Actors the balancer can be applied to). Consider a load balancer which asks Actors for some cost metric. This might include their willingness to move (an Actor may have a particular affinity with a local resource), their size (how much state they are carrying with them) and other such attributes. Balanced Actors (Actors which are to be load balanced) must therefore be capable of generating such metrics and the 'Decision Actors' must now process more information and make a more complicated decision. An ideal solution must balance computational cost and invasiveness against the results achieved.

5.4 Finding Actors

Many Actor and agent languages place restrictions on Actor/agent mobility. GLINT, for example, insists that for an Actor to migrate only one other Actor may know its name. Furthermore GLINT Actors must give up their acquaintances upon migration. From the outset STAGE's migration model was designed to be as unrestricted as possible whilst minimising, in Sackman's words, the 'under the bonnet magic'.This section discusses STAGE's migration model and its relationship to other languages presented.

To fully support the Actor model Actors must be free to communicate with any acquaintances they have acquired regardless of the current location of such acquaintances. In an Actor language without distribution where inter-Actor communication is via local message passing this is trivial. In this scenario the location of each of the Actors' mailboxes can be stored in a suitable data structure within the execution environment. When an Actor sends a message the local mailbox is located and the message placed within the mailbox.

5.4.1 Location based naming

Many Actor languages also allow distribution. ERLANG is one such language and provides distribution through its `remote_spawn` primitive. For this section we consider distribution as the ability to send messages between Actors on potentially remote hosts, precluding mobility. This adds extra complexity to message sending. We must now consider network errors and location (where to send the message to). The location problem is simplified by virtue of the fact that the only way to interact with an Actor is by sending it a message. Furthermore to send a message to an Actor the target Actor must be an acquaintance of the originating Actor. From the definition of an acquaintance we conclude that the originator must know the name of the target. This leads us to a simple solution to the location problem which parallels that of email addresses. This solution proceeds by naming each Actor by the concatenation of both its unique name (in the pure Actor sense) and its location. When an Actor sends a message it splits the target Actor's name into its constituent parts. It can then send a message addressed to the Actor to the relevant remote address (figure 5.4). Since we assume Actors cannot migrate it is valid to assume that the location will be valid until the Actor's death. This scheme is employed by the logical agent language QUPROLOG [3] whereby Actors (called simply threads in QUPROLOG nomenclature) are referenced by names of the following form:

```
thread:process@hostname
```

In a large distributed system locating Actors by a specific thread identifier within a process is clearly undesirable. However, due to the flexible nature of Prolog systems (such as QUPROLOG), higher-level naming constructs can be easily added.

[3] See http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html

Source Actor name	Target Actor name	Source action
john@vertex10.doc.ic.ac.uk	will@vertex10.doc.ic.ac.uk	local_send(will, msg)
john@vertex10.doc.ic.ac.uk	james@amps.ee.ic.ac.uk	remote_send(james, amps.iee.ic.ac.uk, msg)
john@vertex10.doc.ic.ac.uk	neil@cluster1.google.com	remote_send(neil, cluster1.google.com, msg)

Figure 5.4: A naming scheme for Actors with fixed locations.

5.4.2 Finding mobile Actors

Locating Actors which are capable of inter-host migration precludes the techniques outlined thus far. A more general solution is required, such a solution will necessarily come at some extra cost. A solution to the generalised location problem is required for STAGE. To determine the most suitable solution we consider the following aspects:

- Scalability.
- Complexity.
- Fault tolerance.
- Correctness.
- Speed.

Originating host

In this scheme the host which originated an Actor is considered an authority on the Actor's current location. This is similar to the location based naming of section 5.4.1, with an extra layer of indirection. Each Actor's name does not contain its location (since this will change) rather it contains the location of a host capable of locating the Actor. This scheme will be correct (under the assumption that hosts are honest). A host need only remember the location of the Actors it as originated and a host (in the absence of nefarious activity) will only receive requests for the location of those Actors which were 'born' on the host. Under assumption that no host instantiates a significantly higher than average number of Actors, this scheme is scalable. This scheme also has a low complexity since each host need run no extra components - only the STAGE runtime. However the ability to locate an Actor is entirely dependent on the status of its originating host: if that host fails then the Actor will be lost, and an Actor that cannot be located is no-more useful than a dead Actor. It should also be noted that if the connection between the originating host and other hosts is slow or fails then other hosts will waste time trying to locate the Actor. For these reasons this method provides poor speed and fault-tolerance and was rejected.

Ad-hoc

The simplest, most efficient solution is to provide no support for distributed Actor naming at the language level. Under this regime Actor implementers must ensure that Actors can communicate via a local blackboard, yellow pages directory or are united by local match-maker

services. GLINT is a language which adopts this style, however Actors must loose their acquaintances/collaborators upon migration. In these systems Actors typically do not maintain their collaborators throughout their lifetime and would have to resort to local information gathering at each host they visit. Since STAGE supports migration whilst maintaining references to collaborators this system is unsuitable.

Locators

The most well known, and widely used, system for mapping a logical name to a physical host is the DNS (hostname to IP address resolution) system of the Internet. Under DNS, address resolution is passive. When an agent requires the IP address of a host it makes a request to a DNS server. Importantly a host is not preemptively told when the IP address of domain name changes (as one would expect of a publish-subscribe mechanism). This, along with its hierarchical structure, gives DNS its scalability - a simple request response operation where requests can be serviced by any appropriate node. DNS entries change infrequently and clients can safely cache a result for a period dictated by the expiry time of the DNS record's time to live (TTL).

Actors communicating with other remote Actors require the current location of the Actor with whom they wish to communicate for every message they send. If we restrict ourselves to a DNS-style (passive) solution we have two options: eager and lazy lookup.

Eager lookup determines the location of an Actor before each send. This has the obvious disadvantage is that if an Actor sends a large number of messages address lookup soon becomes a performance choke point. The main advantage of this approach is that there is no need for the individual theatres save any information pertaining to the global situation such as the location of other Actors: locations are looked up before each use.

A lazy approach encourages theatres to look up the location of an Actor once, store this locally, then assume that the Actor has not moved. This raises an obvious question: what happens when an Actor moves? since this renders earlier lookups invalid.

STAGE's solution

STAGE's solution to the location problem aims to do the simplest thing that could possibly work. STAGE takes the view that location mechanisms are 'hints' or 'indications' of the location of Actors. To give an informal definition we consider the real-world analogy of a human in a city looking for the post office. Of course, the human may choose an authority (such as a tourist information office) but in the absence of such an authority he engages in a social iteration with his peers. He may ask his peers for the location of the post office. Some may be certain, others unsure but have a rough idea and others may be deliberately misleading. Regardless of the result he can evaluate the responses and chooses one based on some criteria. He may be successful and find the post office at the location specified, however if he doesn't he can ask other peers. This analogy highlights three important aspects that define STAGE's approach - the ability to try, to verify and to retry if required.

To realise this in STAGE, theatres can say 'no' to a message if the Actor is not present. An ex-

```
Actor B on theatre T1
Actor A lookup Actor B
Actor A told Actor B on theatre T1
Actor B moves theatre T2
Actor A messages Actor B on theatre T1
theatre T1 says NO
Actor A retries
```

Figure 5.5: A theatre says 'NO'.

ample of this style of integration is shown in figure 5.5. Because locators are just hints we can add as many as required - allowing scalability, fault tolerance and speed. STAGE currently only supports hints from a single custom location component named a 'StageManager'. The implementation provides extension points should Actors wish to accept hints from other sources such as DNS servers, text files, auto-discovery systems such as Apple's Bonjour service and so forth.

A pleasing consequence of STAGE's try-reject-retry cycle is that caching can be added to support lazy lookup without fear of error since if entries in the cache are incorrect the target theatre will reject messages. If a message is rejected the cache entry is purged and further hints are solicited.

5.4.3 Driver Actors

STAGE allows Actor bodies to include any valid PYTHON statements. It is not unusual to permit statements from the underlying language in Actor definitions. SALSA Actor definitions can include JAVA statements and SCALA Actor definitions can include SCALA statements. If blocking calls or thread control operations are performed by an Actor then the Actor may not exhibit the behaviour the implementer may expect, especially under migration. Furthermore, for the Actor model to be beneficial all inter-Actor communication must be via the method-call (message passing) abstraction provided by STAGE. If an Actor implementer is forced to resort to providing extra communication channels ('side channels') or threads to achieve the desired behaviour then it demonstrates a deficiency in STAGE's features.

Many existing Actor languages describe only Actor-Actor interactions. Motivated by the slogan 'Everything is an Actor' they avoid explicitly dealing with I/O; instead choosing to represent the 'user' or 'file' as an Actor. This is under the implicit assumption that underlying Operating System or execution environment presents such abstractions and in reality they do not. STAGE aims to provide a usable and complete Actor language and as such cannot simply assume that a 'user Actor' generates messages representing the user's input; it must provide concrete (PYTHON) implementations of such Actors. In STAGE's nomenclature these are known as 'Driver Actors'.

STAGE provides a number of Driver Actors. These include message-generation Actors (for timing), file and terminal I/O Actors and graphical widget Actors. Since PYTHON provides lightweight support for common file operations and terminal input any many of these Actors simply wrap the relevant PYTHON statements. They wrap the statements in a manner consistent with the underlying theatre implementation which ensures that STAGE theatres behave as

expected.

But how do existing PYTHON frameworks interact with STAGE? Firstly native PYTHON code may only communicate with Driver Actors (this is not currently enforced by the runtime). It is the role of Driver Actors to forward messages to 'normal' STAGE Actors. To communicate with Driver Actors PYTHON code must use the STAGE Native Interface. The interface is kept deliberately simple and consists of a `send` operation and a `send_receive` operation. The `send` operation attempts to deliver a message to the relevant Actor, it returns as soon as the message has been placed in the relevant mailbox. The `send` primitive returns a status value which specifies if the send was successful. The `send_receive` operation sends the specified message and waits for a reply. A `receive` primitive is not included since it is not compatible with the Actor model. It is not compatible because native code is not an Actor and does not have an Actor name. Actors cannot send messages to code without a name. The `send_receive` primitive overcomes this restriction by generating a worker Actor which handles the send and receive on behalf of the native PYTHON code.

'One-way Drivers' either receive or generate messages but not both. Such Drivers are simple to design and are less prone to error since they cannot generate circular waits. The most complex drivers are those which require bi-directional communication. These tightly couple the STAGE execution environment to the Device. To demonstrate the problems associated with bi-directional native communication the following section discusses graphic user interface elements ('widgets').

GUI Actors

A section has been devoted to graphical user interface Actors (GUI Actors) because they were an important design decision and one that other Actor languages must face if the wish to allow the development of 'real' desktop applications.

The first design decision was to determine the best GUI library to incorporate into STAGE. Incorporating one library does not preclude the inclusion of other libraries, but it does encourage a standard methodology. The GTK+[4] library was chosen since it has well-established PYTHON bindings via the PyGTK API. The problem remains to integrate STAGE's Actor model with GTK's threading and interaction model. The most striking difference between the two models is that the Actor model uses copying and messaging whereas PyGTK uses method calls and shared state.

The eventual solution is shown graphically in figure 5.6. The methodology is best explained by example. Consider an Actor which represents a button - a Button Actor. Clearly when such an Actor is on a host it should appear to the user as a button on the screen. We designate the term 'physical extent' to describe the externally visible behaviour or appearance of an Actor. Our Button Actor's external extent is a set of pixels on the screen. PyGTK provides an object which represents a button on a screen, namely a gtk.Button. However this is a passive object and is not an Actor, it does not accept STAGE's Actor messages nor can it migrate since it depends on local static data. More generally we cannot migrate device code since we may not have the source, or the device executable may be binary. For these reasons we cannot simply use a gtk.Button as the Button Actor we require.

[4] See http://www.gtk.org/

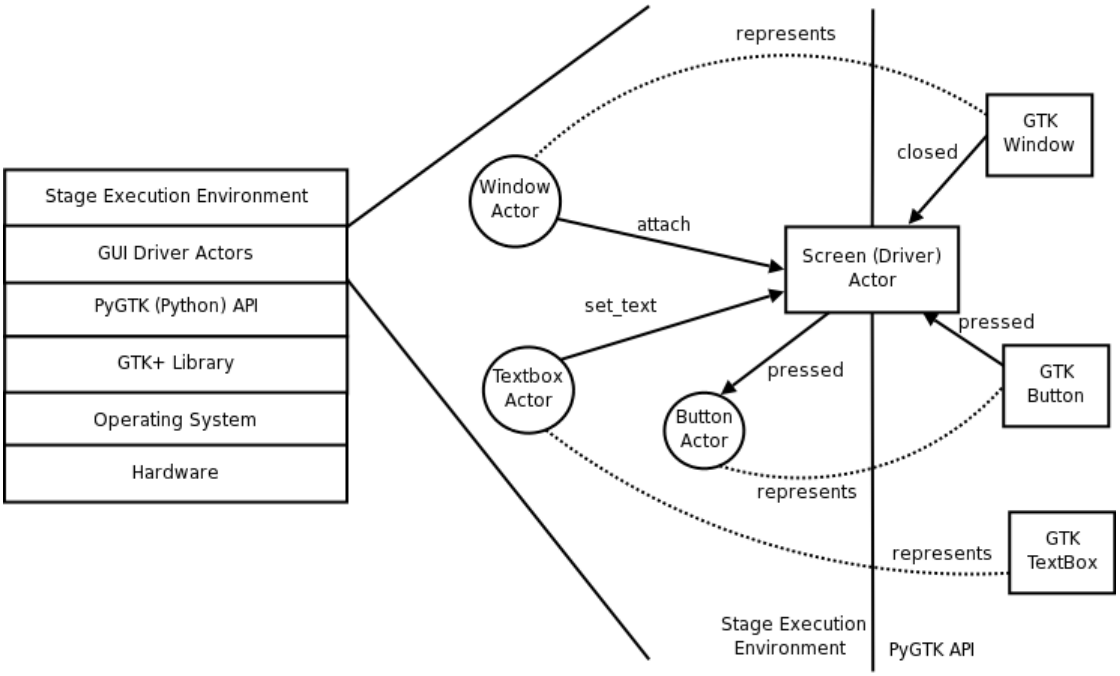

Figure 5.6: GUI drivers in STAGE.

To clarify the problem we distinguish the Actor (which conceptually represents the GUI element) and the physical extent. For our Button we have both a Button Actor written in STAGE (the conceptual button) and the gtk.Button (PYTHON/GTK) object (the physical extent). We require a bidirectional link between the two elements - the GUI must tell the Actor when the button is clicked, and the Actor must tell the GUI what text to display on the button. We cannot use PYTHON references since messages are passed by copying, rendering references invalid. Similarly such references (if possible) would not be valid after migration.

To solve these problems We use a system of attach/detach calls. The Actor and the GUI do not posses direct references to each other rather a third party maintains the association. This Actor is called the `Screen` Actor. When an Actor (such as a Button Actor) is born or migrates to a theatre it informs the local screen that it exists via an 'attach' call. When the `Screen` receives such a call it generates the physical extent of the Actor using PyGTK and maintains a link between the two entities. When a button (in the GTK sense) is clicked it informs the `Screen` Actor using the native interface that a click has occurred. The `Screen` Actor then informs the STAGE Button Actor that a click has been performed and the Button Actor performs the relevant behaviour. Similarly if the Button Actor wishes to change its text it informs the `Screen` which in-turn informs the gtk.Button. When an Actor wishes to leave a theatre (or dies) it informs the `Screen` using a 'detach' message which removes its physical extent. In some sense the `Screen` provides a gateway between the two paradigms. One final implementation detail is that a button cannot exist on a screen in a vacuum - GTK only allows elements to exist on Windows or sub containers that eventually exist in Windows. This means that a button cannot migrate in isolation since it would not be visible when it arrived at the new host. To prevent confusion STAGE considers the Window as the unit of migration for GUIs. Further work could allow

Actor	Type	Description
Window	Container	Outermost container for GUI elements
Button	Input	Generates an event when pressed
TextBox	Input	Accepts a line of user input
HtmlView	Rendering	Renders HTML
Label	Output	Displays a line of text
Canvas	Drawing	Container for vector graphics
Rect	Drawing	Represents a rectangle on a canvas
Line	Drawing	Represents a vector line on a canvas
HList	Layout container	Displays its children horizontally
VList	Layout container	Displays its children vertically

Figure 5.7: A subset of the GUI widgets available in STAGE.

the migration of individual elements by creating temporary Windows on hosts for orphaned elements. A small selection of the Actors available is given in figure 5.7.

5.5 Errors, failures & reliability

5.5.1 Actor error

Many Actor-based languages and multi-agent systems provide little or poor support for error handling and recovery. One such example is this project's forerunner, GLINT, which has no concept of an error and thus no error handling is provided. Behaviour in the presence of errors is underspecified. In GLINT calling methods with the wrong number or incorrectly typed parameters causes unpredictable behaviour. Similarly making a method call on a dead Actor results in an infinite wait.

ERLANG, which was designed for long running, resilient systems, provides both a try/throw/-catch mechanism and a supervisory model (discussed in section 5.5.2). Error handling in Actor/agent systems is non-trivial due to the autonomous/independent nature of Actors. It is therefore unsurprising that many languages of this nature make no attempt to deal with exceptional behaviour. MADKIT [4], a multi-agent language in JAVA, is one such language. In fact MADKIT agents naively inherit the JAVA exception model and as an undesirable consequence, uncaught exceptions can propagate up the call stack resulting in the termination of an agent's thread [72].

Listing 40 Handling errors in PYTHON.

```
1  try:
2    f = open('/tmp/somefile')
3    ...
4  except IOError:
5    # handle the error
6    ...
```

PYTHON, uses a raise/try/except (listing 40) pattern to handle errors. This works well for typi-

cal PYTHON code, where errors are handled on a statement-by-statement basis. In a distributed and mobile Actor model almost every statement can be the source of an error. Line-by-line error handing soon becomes redundant, tedious and error prone. For this reason STAGE uses a declarative error handling system. Errors are broken down into error types (figure 5.8) and an Actor's behaviour is specified for each error type. This decision was taken as each Actor should perform a small set of actions (a 'role' in a society to use Hewitt's terminology) which makes policy based error handling more attractive.

- Communication error - A message failed to be delivered.
- Actor location error - Unable to locate an Actor.
- Response timed out - The target Actor did not respond within the predefined time limit.
- Invalid request - The Actor did not understand the request.
- Request failed - The target Actor understood the message but raised an exception. Both the caller and callee are informed of the error.

Figure 5.8: Error types in STAGE

5.5.2 Supervision trees

ERLANG provides a higher-level abstraction for error handling and recovery called a supervision tree [3]. In this model processes are split into workers and supervisors. The behaviour is analogous to its human counterparts: workers perform the actual work and supervisors monitor workers and other subordinate supervisors. This forms a supervision hierarchy. An error raised by a child worker or supervisor is reported to its parent supervisor which can take appropriate action. Figure 5.9 shows how a supervision tree can be used to model a real-world scenario.

STAGE provides supervision functionality which mirrors that of ERLANG. However STAGE provides supervision trees as core libraries rather than as a language feature. STAGE has the advantage that workers and supervisors can migrate between hosts - ERLANG does not provide process migration. To capitalise on the migration capabilities of the language STAGE splits workers into two categories: mobile workers and fixed workers. Fixed workers are workers which are relying on a local resource and thus cannot migrate. Mobile workers are workers which are not tied to a local resource (or are willing to accept a different local resource). In ERLANG a supervisor can kill or restart its children as dictated by a supervisory strategy. In STAGE a supervisor can restart or kill its children (as per ERLANG) but may also migrate its mobile children if they are erroring or performing badly on a specific host. Furthermore supervisors are informed when their children migrate and may choose to revert the migration.

5.5.3 Theatre closure

STAGE has been designed to allow computation to continue whilst the underlying network of hosts changes dynamically. The addition of a new theatre poses no problems. A new theatre

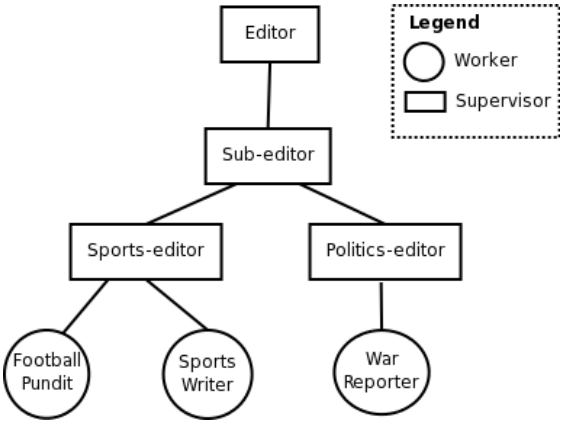

Figure 5.9: A supervision tree for a national newspaper.

announces its willingness to accept Actors and Actors may choose, or be told, to take advantage of a new host by migrating or interacting with it. The removal of a theatre requires greater support since we must ensure that any Actors currently resident in the theatre safely transition from the host to a new location.

The implementation focuses on the following conflicting goals:

1. Actors which wish to stay alive are able to migrate before complete termination of the theatre occurs.
2. The theatre is able to shutdown quickly.

Requirement one is motivated by our desire to support long-running reliable computation, the loss of an Actor may represent the loss of a partially computed result. Requirement two is motivated by real-world constraints, namely once a host begins shutting down or failing, processes (such as a STAGE theatre) have a short period of time two complete their operations before the CPU halts. For example a rack of hosts supported by an uninterruptible power supply (UPS) usually have around 30 seconds of life before complete loss of power. STAGE makes no attempt (at the language level) to deal with ungraceful failure of a theatre caused by catastrophic failure of a host. This does not forbid or prevent the addition of extra features in-language (such as Actor replication and polling) to deal which such failures. Moreover many failures are not characterised by such sudden stops. These are termed 'forecasted node failures' [77] and include power failure (with a small UPS), disk failure, critically low disk space and error indicators such as rising temperature. In all of these scenarios Actors are able to escape the failing host.

STAGE uses message passing to inform each Actor within the closing theatre that theatre is shutting down. To respond to this message Actors must implement the `theatre_closing` method which will be executed upon termination of the theatre. The valid responses to such a message are:

1. Ignore the message - The Actor will die with the theatre.

2. Migrate to a host known to the Actor - The Actor will survive if the remote host is operational.

3. Request to be migrated a theatre of the local theatre's choice - The Actor will remain alive but may be separated from its collaborators.

4. Interaction with another Actor - This action can be combined with any of the preceding actions.

Responses one and two are relatively straightforward. Response three may result in a sub-optimal configuration of Actors to hosts. STAGE does not attempt to perform an optimal or even good distribution of Actors since requirement two states that the theatre should shut-down quickly - collaborating to select optimal theatres to migrate to would unduly delay the termination of a theatre. However once settled in a new theatre Actors may collaborate and migrate to form a better configuration. An interesting implementation detail is that messages pertaining to the closure of a theatre must be given priority in an Actor's message queue. If an Actor wastes vital seconds dealing with other requests it may miss its opportunity to migrate and will die with the closing host.

5.6 Stage as a language embedded in Python

Embedding a language in an existing language allows a more complete and usable language to be developed. An established language, such as PYTHON, has a large and active developer base. These developers are all potential STAGE developers. The motivation for keeping the syntax as close to PYTHON as possible (with no rewriting) is to maintain familiarity for PYTHON developers and to preserve existing tool support. The importance of tool support should not be underestimated. The prevalence and popularity of JAVA can be attributed, in no small part, to the excellent ECLIPSE IDE[5].

PYTHON was chosen after preliminary prototyping. JAVA was considered but proved too verbose: to follow JAVA's conventions Actor definitions would require both an implementation and an interface. PYTHON is dynamically typed and supports duck typing which is characterised by the expression, 'If it looks like a duck and quacks like a duck, it must be a duck.' [73]. Such typing allows STAGE programs to form large open systems: system-wide types are not required rather the Actors simply provide capabilities (as methods). The lack of static type information also reduces the textual and conceptual size of programs.

PYTHON has language support for a number of commonly-used datatypes such as lists, dictionaries and tuples. These structures encourage fast prototyping. Consider a model of a set of exam marks: in PYTHON a programmer may initially represent this model as a list of name-value pairs: `[('john', 5), ('neil', 10) ...]` whereas in other languages extra objects such as `Mark`, `Student` and `ExamMarks` may be required. Since PYTHON is an Object Oriented language it is possible to create such objects however it is a necessity that in a language designed for fast prototyping that a simple solution (the list of pairs) can be explored before determining if more explicit domain abstractions are required.

[5] See http://www.eclipse.org/.

The STAGE language exhibits weak mobility as defined in [20]. This means that although execution after migration will not continue from the point in the code where the migration was initiated an Actor will be notified that it has been moved (and will be able to take relevant actions). This is termed 'weak mobility'. Since STAGE is an embedded DSL (or 'internal DSL' to quote Martin Fowler [31]) weak mobility has been selected as it fits better with PYTHON's code and data model than its strong mobility counterpart. With support for specifying code to be executed on arrival at a new host, weak mobility is sufficient to encode complex migrating Actors.

5.6.1 Threading and the GIL

PYTHON provides a threading module [5] which contains threading operations similar to those found in JAVA. The distinction is that JAVA provides these operations at the language level whereas PYTHON provides such operations as a module. PYTHON threads are Operating System level threads, which confers the benefit that they have the opportunity to be scheduled on different cores of a multiple core CPU. However there is a high switching overhead since thread operations inevitably involve a context switch.

The global interpreter lock or 'GIL' is a lock found in the PYTHON interpreter which protects the interpreter's internal structures from corruption caused by concurrent access. The visible effect of this design decision on PYTHON programs is that no two PYTHON bytecodes are executed at exactly at the same time. From a performance perspective this is undesirable since although we can thread our programs (and those threads may be scheduled across multiple cores) only one will be active (holding the lock) at any one instant[6]. This problem is not unique to STAGE and has been encountered before [63].

To overcome this problem hosts with multiple CPUs should create one theatre per CPU core. This avoids the problems associated with the GIL since the GIL only locks PYTHON bytecodes in the same interpreter instance. This is a little undesirable since this means Actors must migrate across cores themselves (although this can be done automatically using one of the load balancing libraries provided with STAGE). Modifying the STAGE execution environment to automatically spawn further PYTHON interpreters to avoid this problem would be beneficial further work.

[6] This is not strictly true since operations (such as I/O) may drop the lock however non-blocking pure PYTHON code exhibits this behaviour.

Chapter 6

Evaluation

Evaluation of a new language, such as STAGE, is a difficult task since it does not have the user-base and abundance of examples that more established languages enjoy. During development of the language I have generated many examples, ranging from large distributed and graphical applications to smaller tests which demonstrate specific features. This evaluation considers:

- **Language** - The quality, clarity, conciseness and development time of STAGE programs.
- **Distribution** - The extent to which STAGE supports the development of large, scalable, reliable and evolutionary distributed systems.
- **Performance** - The runtime of STAGE programs.

6.1 Language

This section concerns an evaluation of the language's clarity and conciseness. Both criteria are inherently subjective for which it is difficult, if not impossible, to obtain objective measurements. To evaluate this aspect of the project we compare a set of small examples and their implementations in a selection of languages. To add structure to the evaluation of the examples I will address the following issues:

- **Overall size** - This is probably the most contentious metric as word counts and semi-colon counts are widely regarded as 'meaningless metrics'. I will give quantitative token counts but will also present an overall assessment of the size. The language is to support fast prototyping of distributed systems and thus we require conceptually small programs.
- **Development time** - One of STAGE's core aims is to promote simple and fast development of complex distributed systems. The main criterion will be that it should be possible to create complex systems quickly. A measurement of programming time under controlled conditions will not be required rather an overall impression of the time and cognitive ability required to generate the examples will be sufficient. The number of times and the

length of time spent finding and reading help pages and language documentation will also be noted.

- **Scope for error & complexity** - Errors in concurrent programs are notoriously difficult to track down. The potential for ambiguous and unclear programs will be assessed. I have not deliberately searched for such programs however any problematic code experienced whilst writing STAGE programs is noted.

6.1.1 Examples

The following section demonstrates STAGE features through short examples. These examples have been stripped down to their minimum functionality to ensure that STAGE's features are clearly visible.

Chat system

In this example we implement a simple chat system. Users may enter and leave the network as they wish and may type messages which are sent to all other willing participants in the network. Listing 41 shows a STAGE implementation of this system and listing 42 shows the same system in JAVA. In the STAGE example the ChatServer Actor is of type 'BServer' this is simply a 'bidirectional server' whereby the server maintains a reference to all of its clients and its clients know the unique name of the server. Readers should note that my JAVA implementation has been written in a manner which allows it to fit onto one page; it is not meant as comprehensive Object Orientated treatment of the problem. By comparing and contrasting the two implementations we observe the following.

- Lines of code - The STAGE solution occupies 18 lines whereas the JAVA solution spans 94 lines of code.
- Naming - In the JAVA example to locate the server we must use a URL and port combination. STAGE abstracts up to logical Actor names. This means we can simply refer to the 'ChatServer', which is a location independent identifier, and the STAGE runtime takes care of resolution.
- Reliability - The JAVA solution is unreliable since if the host on which the chat server is running leaves the network then all the clients fail. If a client decided, upon failure of the server, that it should become the server, then this would be of no use since all the clients refer to the server by URL. Moreover the code to make this decision is not included in the code presented and would likely be larger than the existing code. In STAGE if the host currently executing the chat server wishes to leave the network, Actors (including the chat server) can migrate away from the closing host. In this scenario the system continues without disruption.
- Explicit threading and synchronisation - The STAGE solution contains no keywords pertaining to synchronisation or threading. Synchronisation and thread creation is scattered throughout the JAVA code. Looking at the JAVA client we note that the program must

spawn two threads: one to listen to messages from the console and one to listen to messages from the server[1]. Similarly we must explicitly synchronise access to the list of clients in the server to prevent concurrent access, although we could make use of JAVA's synchronised collections found in the `java.util` package.

- Explicit startup - When running the JAVA system users must explicitly start a server instance on the relevant host. STAGE uses lazy instantiation of Singleton Actors which means the server is started automatically when a request is made to it.
- Persistency of the connection - The JAVA system uses persistent connections to transfer text to and from the clients. If the connection fails at any point then the client fails. This represents tight (runtime) coupling between server and client. STAGE Actors only connect to remote theatres to transfer messages and use a retry strategy (defined by the theatre) to deliver messages in the presence of failure. For this reason the STAGE system is more suited to dynamic, mobile environments.

This small example demonstrates that STAGE has distribution and concurrency at its core. It shows that STAGE programs are small and development time is small (the example took no more than 30 minutes to write and test). It shows that other languages (in this case JAVA) prevent fast prototyping of distributed applications since they require knowledge of a vast library of interfaces and tools. Looking at the JAVA code notice how many different objects are used (`Readers`, `InputStreams`, etc) and note that do not directly contribute to logic of the solution. Readers should not presume that JAVA is, in any sense, not suitable for the development of such systems. JAVA (especially with the inclusion of JAVA RMI) can create large distributed systems, the distinction is that they can be developed quickly and more clearly in STAGE.

Listing 41 A simple chat system in STAGE.

```
1  class ChatServer(BServer):
2
3    def say(self, message):
4      for child in self.children:
5        child(message)
6
7  class User(MobileActor):
8
9    def birth(self):
10     self.server = ChatServer()
11     self.server.addchild(self.incoming)
12     Keyboard().listen(self.typed)
13
14   def typed(self, msg):
15     self.server.say(msg)
16
17   def incoming(self, msg):
18     print ">", msg
```

Network monitoring

Consider a set of hosts each running the STAGE interpreter. Figure 6.1 shows how we can use Actors that move freely (although we may wish to provide some security) to a host to warn a

[1] Another solution involves non-blocking I/O, however this style is not indicative of most existing JAVA programs.

Listing 42 A 'simple' chat system in JAVA.

```
1  public class ChatServer {
2
3    public static final int PORT_NUMBER = 7878;
4
5    private Collection<Chatter> chatters = new LinkedList<Chatter>();
6
7    private void startServer() throws IOException {
8      ServerSocket s = new ServerSocket(PORT_NUMBER);
9      try {
10       while (true) {
11         Socket socket = s.accept();
12         Chatter chatter = new Chatter(this, socket);
13         chatters.add(chatter);
14         chatter.start();
15       }
16     } catch (Exception e) {
17       System.out.println(e);
18     }
19   }
20
21   synchronized void tellAll(String s) {
22     for (Chatter chatter : chatters) {
23       chatter.writeToClient(s);
24     }
25   }
26
27   public static void main(String[] args) throws IOException {
28     new ChatServer().startServer();
29   }
30 }
31
32 class Chatter extends Thread {
33
34   private BufferedReader in;
35   private PrintStream out;
36   private ChatServer server;
37
38   public Chatter(ChatServer server, Socket client) throws Exception {
39     this.in = new BufferedReader(new InputStreamReader(client.getInputStream()));
40     this.out = new PrintStream(client.getOutputStream());
41     this.server = server;
42   }
43
44   void writeToClient(String s) {
45     out.println(s);
46   }
47
48   @Override
49   public void run() {
50     try {
51       while (true) {
52         String line = in.readLine();
53         server.tellAll(line);
54       }
55     } catch (IOException e) {
56       /* Ignore */
57     }
58   }
59 }
60
61 public class ChatClient {
62
63   public static void main(String[] args) throws IOException {
64
65     Socket socket = new Socket("shell1.doc.ic.ac.uk", ChatServer.PORT_NUMBER);
66
67     final PrintWriter serverOut = new PrintWriter(socket.getOutputStream(), true);
68     final BufferedReader serverIn = new BufferedReader(new InputStreamReader(socket.getInputStream()));
69     final BufferedReader consoleIn = new BufferedReader(new InputStreamReader(System.in));
70
71     new Thread(new Runnable() {
72       public void run() {
73         while(true) {
74           try {
75             System.out.println(serverIn.readLine());
76           } catch (IOException e) {
77             /* Ignore */
78           }
79         }
80       }}).start();
81
82     new Thread(new Runnable() {
83       public void run() {
84         while(true) {
85           try {
86             serverOut.println(consoleIn.readLine());
87           } catch (IOException e) {
88             /* Ignore */
89           }
90         }
91       }}).start();
92   }
93 }
```

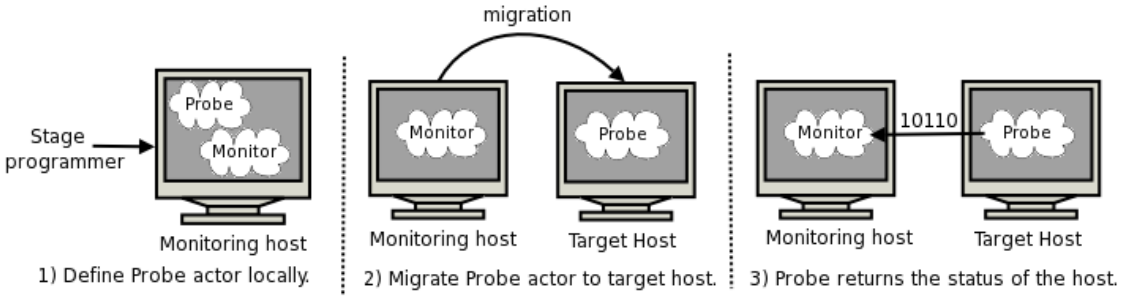

Figure 6.1: A scheme for monitoring hosts with Actors.

central server of impending problems (low disk space etc) or nefarious activity (network card in promiscuous mode etc.). The advantage of this approach over traditional tools is that each host has no persistent code (the presence of such code can lead to versioning problems). The communication overhead is reduced since the monitoring is performed in place. The probe does not constantly transmit data, it monitors and aggregates data autonomously and only sends data when defined by the monitoring strategy. Finally we can migrate arbitrary probes or monitors, we may migrate a more heavyweight probe to a host when a lightweight probe detects unusual behaviour. A more complete overview of the potential for agent/Actor based network maintenance is found in [16]. Listing 43 demonstrates the simplicity of migration in

Listing 43 Probing a host in STAGE

```
1  class Probe(MobileActor):
2
3    def birth(self, report, dest)
4      self.dest = dest
5      self.report = report
6      migrate_to(dest)
7
8    def arrived(self):
9      data = self.do_probe()
10     self.report(data)
11
12   def do_probe(self):
13     status = TheatreStatus()
14     return {'Theatre'          : self.dest,
15             'Load average'     : status.loadaverage(),
16             'Operating System' : status.osname(),
17             'Uptime'           : status.uptime()}
```

STAGE. The `arrived` method on line 8 is clear: upon arrival at a host the Actor should perform a local probe and then report back the results. The probe is deliberately over simplistic. Readers should envisage probes of arbitrary complexity. For example a 'learning probe' may spend some time on a known good host and will learn which factors are indicative of a 'correct' or error free host. The probe could then move to another host and if the behaviour of that host is incompatible with the behaviour of a correct host then it raises an alarm or notification.

In a language without migration the probe would have to be installed on each host and configured with the address of the server that it reports to. This is not a significant hurdle, however the problem occurs when we wish to update the behaviour of the probe, or remove the probe completely. Virus checkers must deal with this problem. They require constant updating and patching since the number, type and sophistication of viruses is growing. These packages of-

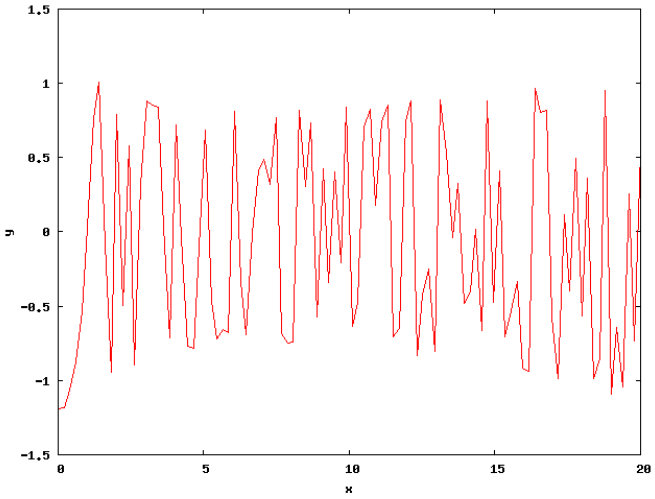

Figure 6.2: Benchmark function integrated using trapezoidal approximation.

ten come with a bespoke update system (which is reasonable since if a virus were to subvert the update mechanism it could render the virus checker useless). However for smaller, more lightweight systems the overhead of having to design and maintain such an update system is a daunting prospect. This is something STAGE provides at the language level through mobility.

6.2 Performance

STAGE's design focuses on creating a language which promotes fast development of complex distributed systems whist providing increased performance though parallelism and distribution. Performance and expressiveness are often in conflict - a feature which increases the expressiveness of the language can often come at the expense of speed. When such decisions arose we favoured expressiveness. STAGE follows the mantra 'Do it, do it right then do it fast'. We compare the performance of STAGE with PYTHON since PYTHON is a lightweight dynamically typed language on which STAGE is based.

6.2.1 Trapezoidal approximation

Trapezoidal approximation is a technique for numerically approximating the definite integral of a function or data set. The formula is given in figure 6.3. The problem can be easily parallelised and is often referred to as 'embarrassingly parallel' since the sub calculations can be evaluated independently. To evaluate the languages we selected a 'complicated' function (plotted in figure 6.2) which comprises of non-trivial operations (cosine, square roots and logarithms). The approximation algorithm was implemented in four languages: STAGE, PYTHON, JAVA and SALSA. The following sections present the performance characteristics of the languages.

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n}(f(x_i) + f(x_{i+1}))\frac{h}{2}$$

where

$$h = \frac{b-a}{2} \text{ and } x_i = (i-1)h + a$$

Figure 6.3: The trapezoidal rule (with thanks to Will Knottenbelt).

6.2.2 Multiple cores

Figure 6.4 shows STAGE's performance using one and two Actors versus a naive PYTHON implementation (the data for this evaluation can be found in appendix A.1). We draw the following conclusions:

- STAGE (one Actor) is not significantly slower than the PYTHON implementation for large amounts of work.
- STAGE (two Actors) exhibits an initial startup cost greater than that of PYTHON.
- STAGE (two Actors) is around twice as fast as PYTHON and STAGE (one Actor) implementations.

From this we can conclude that STAGE's overheads have not adversely affected the performance of PYTHON. Which is desirable since programmers will be unlikely to move from PYTHON to STAGE if the change comes at a significant cost to performance. More pleasingly using multiple Actors STAGE achieves a significant speedup. This is a concrete demonstration that the Actor model can improve performance. At this juncture many readers may ask themselves one of the following two questions:

1. This is an embarrassingly parallel problem which was always likely to achieve a speed-up. This problem is not indicative of the problems that most of the applications being written today have to solve.
2. I could achieve the same results myself by explicitly creating threads and processes and distributing work accordingly.

I concede that both of the above statements are true. However whilst many applications do not consist of large blocks of independent work, they do consist of *some* work which is suitable for parallel execution. Even the most simplistic web applications often display data from different sources - weather information, news, travel information etc. Under the Actor model any potential parallelism, such as retrieving the weather and the news at the same time, will be exploited *by default* whereas one would have to explicitly parallelise an equivalent JAVA implementation.

We have demonstrated that STAGE is able to increase performance on multiple core CPUs and that STAGE's overheads become negligible as work increases. But how does STAGE's performance compare to other Actor languages? If other languages demonstrate the same speed-up do we need another? SALSA (described in section 4.3) is an Actor language which is translated

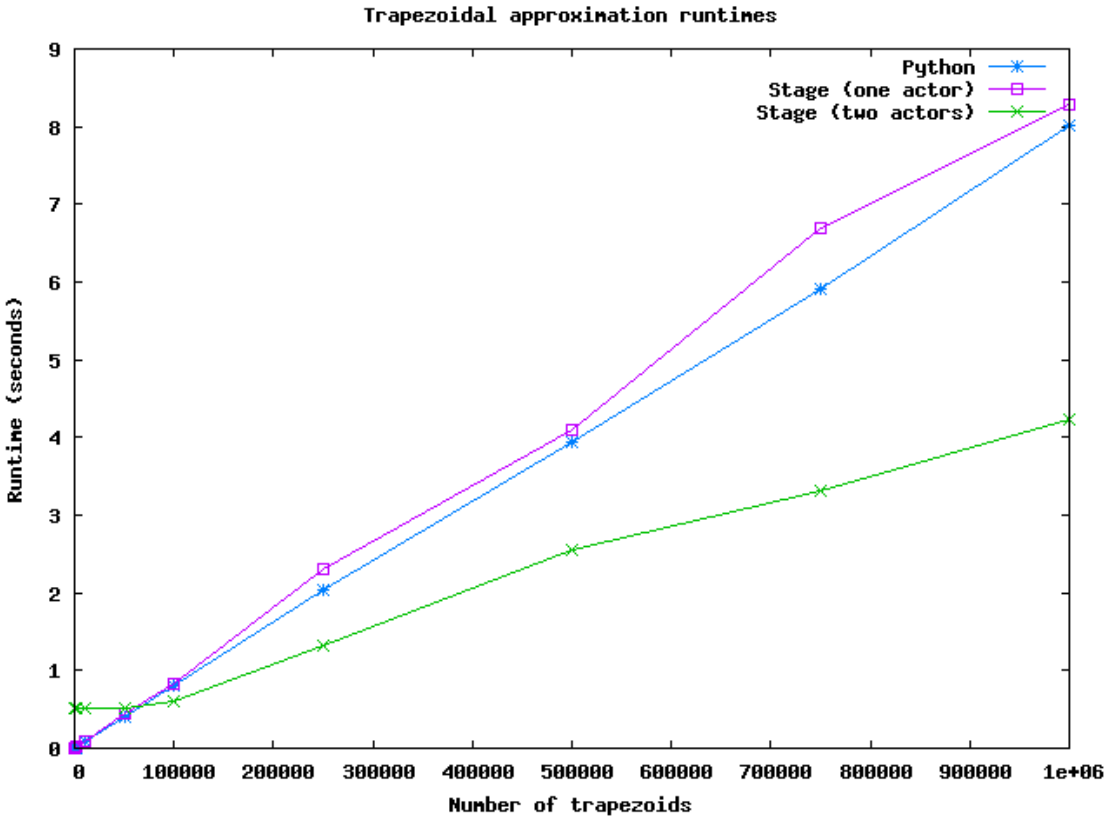

Figure 6.4: Trapezoidal approximation - runtimes on a multiple core CPU.

into JAVA and eventually run on a standard JAVA Virtual Machine. Figure 6.5 shows SALSA's behaviour under load contrasted with that of a naive JAVA implementation and the STAGE (two Actor) implementation. The data for these tests is found in appendix A.1.4. We note the following:

- SALSA is significantly slower than a naive JAVA, PYTHON and STAGE implementation.
- The two Actor (parallel) SALSA implementation is not significantly faster than the one Actor implementation.

I presented these results to the SALSA development team who responded with the following:

> *... All versions of SALSA up to the current one have assumed that it was going to be used distributedly (sic), so there are a lot of checks that need to be done to ensure that messages are going to the correct theater. In distributed cases, the overhead of all the internals is a lot less significant compared to latency over ethernet, etc. so optimizing there hasn't been too much emphasis on performance for non-distributed applications. The main benefit of programming*

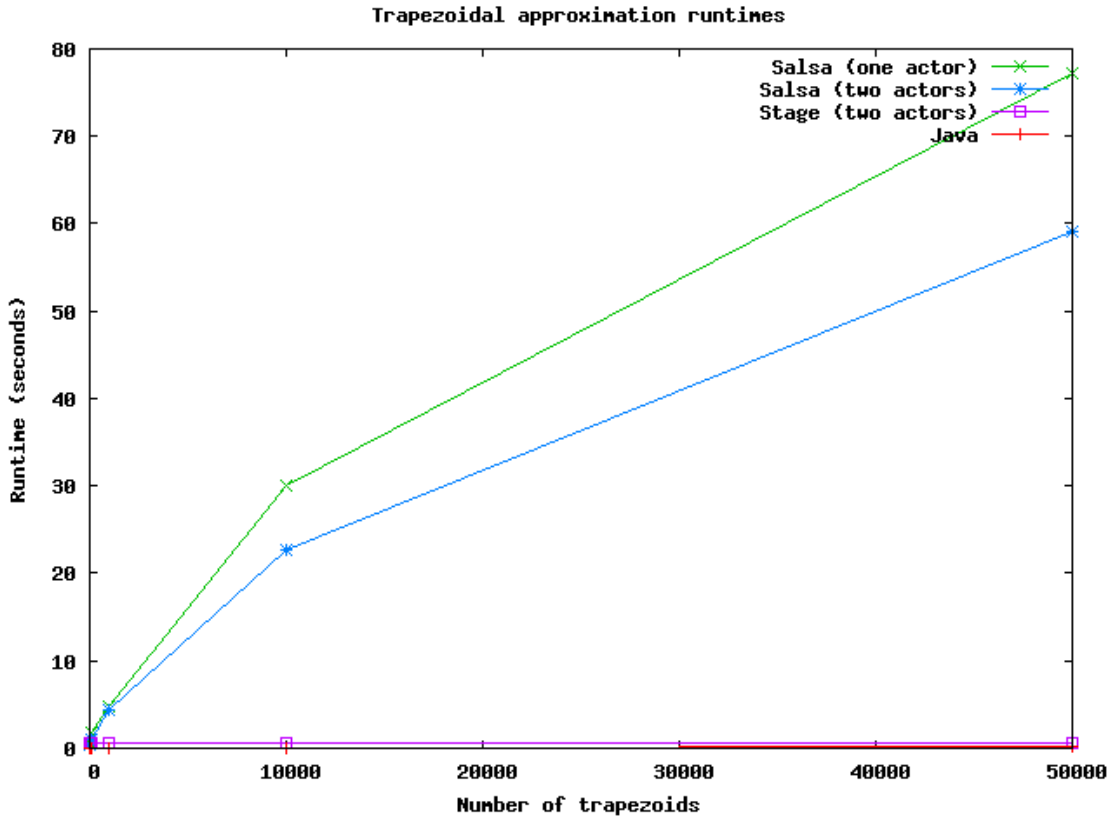

Figure 6.5: Trapezoidal approximation runtimes - SALSA versus JAVA and STAGE.

> *applications in SALSA is that communication between Actors is transparent regardless of where they're located ...* [24]

It would seem that by focusing on distribution the speed of local computation has suffered. This a pitfall that STAGE has avoided.

6.2.3 Distributing for speed

By running the algorithm on one host STAGE's performance was bounded by the performance underlying host. To gain greater speedups we enlist the help of other hosts. Figure 6.6 shows how adding more work-willing hosts to the network reduces the runtime. This algorithm is very sensitive to a slow running host: the result is ready when the slowest host completes its work. This is why we see a small speedup from two to four hosts - one of the extra hosts took longer to produce a result. The overall results are encouraging, and although this task involved little communication, I believe most applications would benefits from the ability to easily spread work across a network of hosts.

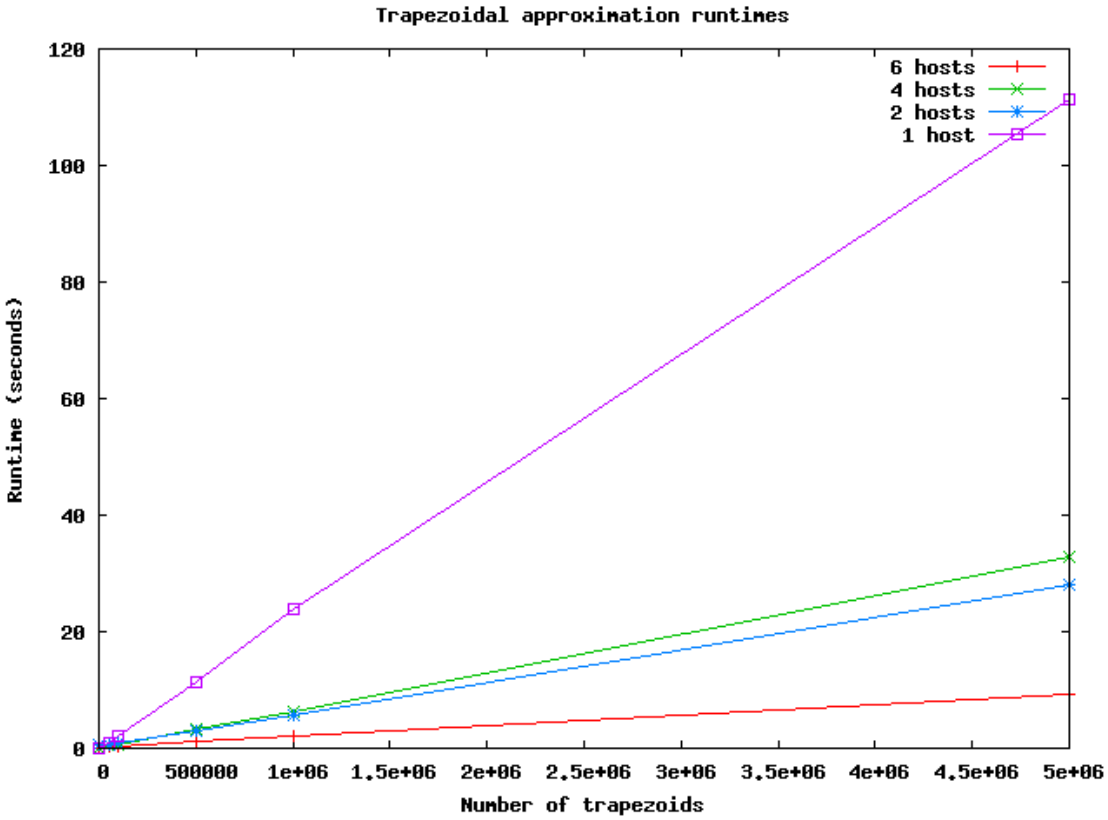

Figure 6.6: Trapezoidal approximation runtime on multiple hosts using STAGE.

6.2.4 The Armstrong challenge

Joe Armstrong (the inventor of ERLANG) sets out his challenge as follows [14]:

- Put N processes in a ring.
- Send a simple message round the ring M times.
- Increase N until the system crashes.
- How long did it take to start the ring?
- How long did it take to send a message?
- When did it crash?

This challenge is deliberately crafted to highlight the benefits of ERLANG's lightweight scheduler. It is a useful test for determining the number of Actors (or processes/threads) that a language can support and the extent to which increasing the number of Actors degrades performance. My criticism of this test is that it does not represent real work. A hypothetical language

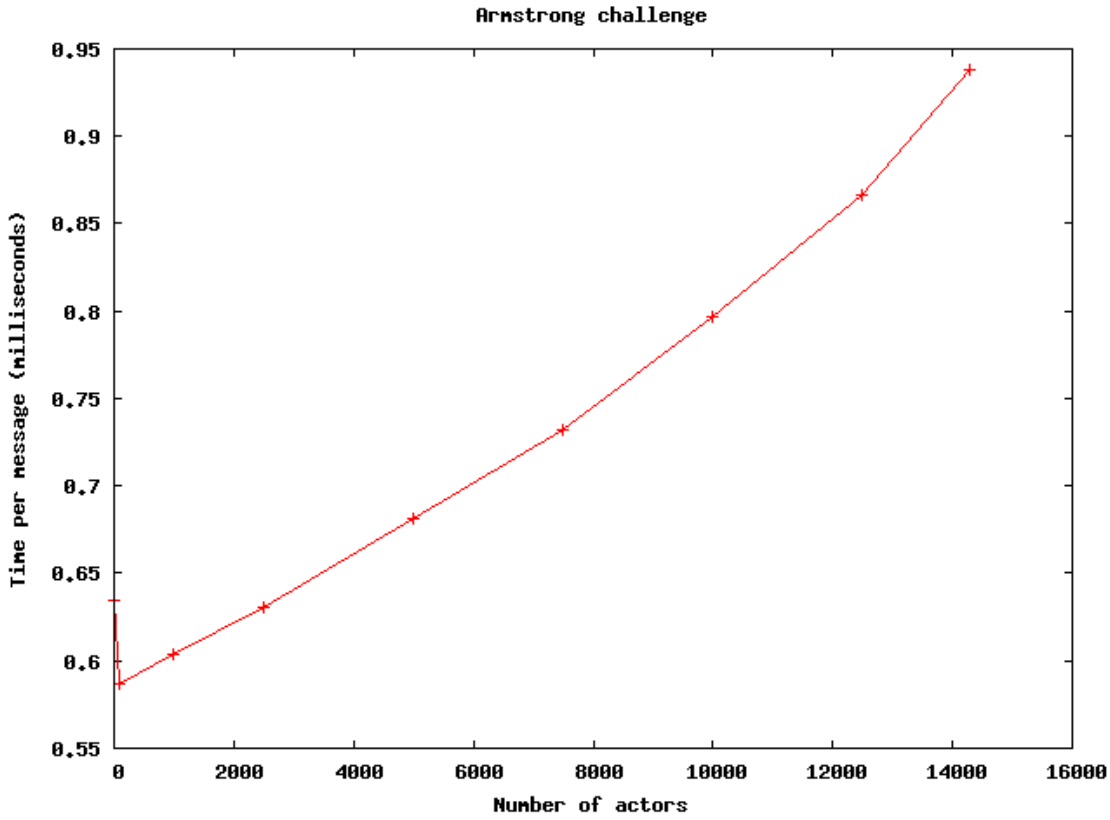

Figure 6.7: STAGE's attempt at the Armstrong challenge.

may support 10^9 Actors, but is having that may Actors useful or desirable? At what granularity should we design our Actors? This test does not attempt to answer these questions but it gives an insight into STAGE's performance under high load.

In my solution to the challenge I create a ring of N nodes (each node represented by an Actor). One node is nominally designated as the 'start node'. The start node is presented with a token which it passes around the ring. Once the token has been passed 100000 times the system halts. The network of Actors is timed from the creation of the first Actor to the final halting state. This means that timings are inclusive of the Actor startup time. The per message time is calculated by dividing the total running time by the number of messages sent. The results are shown in figure 6.7 and the data can be found in appendix B.1.

The results demonstrate the per-message time increases as the number of Actors in the system increases. This is not unexpected since increasing the number of Actors results in a greater number of threads which increases contention and incurs a greater startup time. The results demonstrate is that this overhead is not crippling - the time per message is only increased by 0.3 milliseconds when 14,000 Actors are present in a theatre. 14,000 Actors is greater than one would expect on a host. The Pong game is a relatively large application (see section 6.3.1) and contains around 40 Actors at any one instant. Furthermore since in STAGE messaging is via

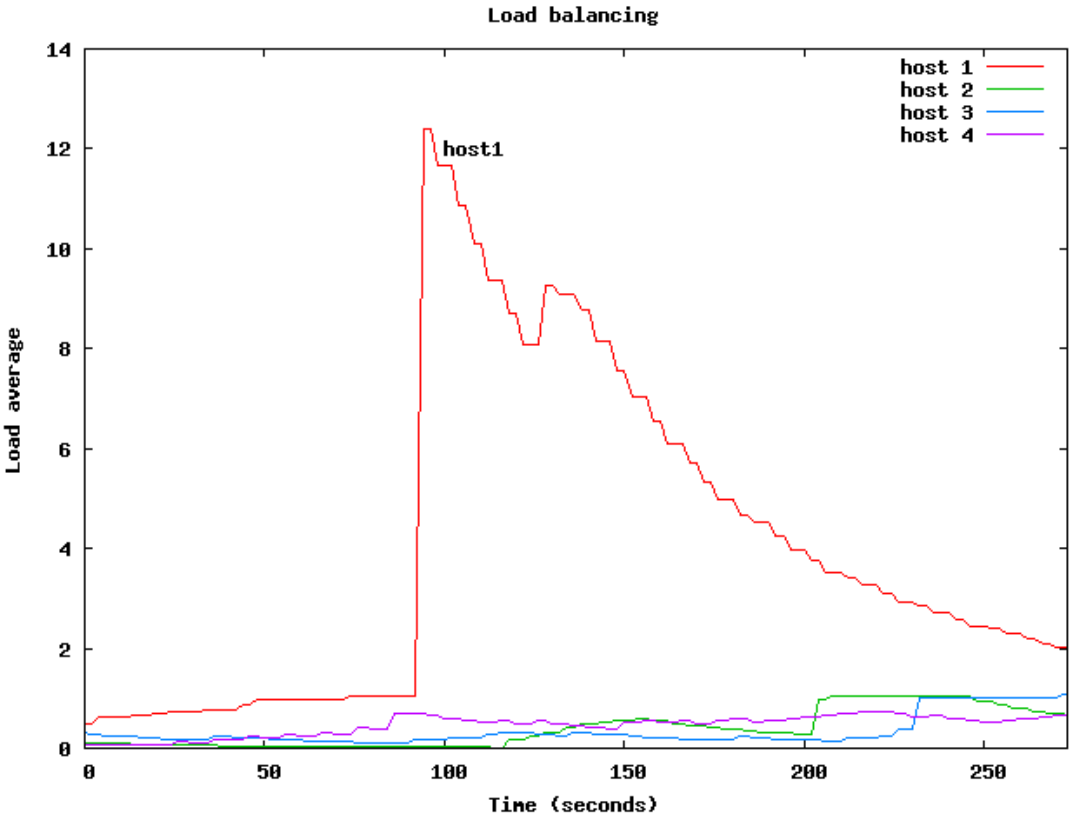

Figure 6.8: Load balancing across four hosts using STAGE.

method call for each token sent an acknowledgement message is sent back, so there are in fact double the number of messages being sent, something I have not included in my results.

6.2.5 Load balancing

The load balancing technique provided by STAGE is given fully in section 5.3.8. To test the load balancer we consider a small network with four hosts. We increase the load on the first host by creating Actors on it. We create a new Actor every 200 milliseconds. Each Actor has a different execution profile and will perform varying amounts of work. This is designed to simulate a web server experiencing an increase in traffic. The results are plotted in figure 6.8 and the raw data is available in appendix C. It is clear from the results that the load balancer is successful in reducing the load on host one. Anecdotally, without load balancing host one became unusable under this load. The load metric used is averaged over a minute - this leads to feedback and over/under balancing. This is the cause of the non-linearity.

6.2.6 Performance conclusion

STAGE's dependence on PYTHON means that STAGE will never be as fast, at least at serial execution, as equivalent C/C++ or assembly language implementations. However STAGE is certainly not slow. By encouraging independent execution and by exploiting multiple core architectures STAGE is able to out perform PYTHON. I have demonstrated that a state-of-the art Actor language is prohibitively slower.

STAGE supports a large number of Actors without prohibitive degradation in messaging times. The exact number of Actors supported by a host is depends on the size and characteristics of the Actors as well as the specifications of the underlying host. Conservative estimates would place the number at 4000 Actors for modest hardware. I have found this to be sufficient for even the most complex systems.

Stage programs are reliable and ensure systems remain operational under error. By ensuring Actors are informed of theatre closure, Actors are able to migrate from terminating hosts. STAGE is completely host-independent - hosts can leave and join a network without disruption.

6.3 Complex distributed systems

To evaluate the language fully it is necessary to program large and complete examples in STAGE. We present two large applications. A realtime distributed game and a distributed web server.

6.3.1 A realtime distributed game

A game was selected since most common desktop programs (email clients, feed readers and browsers) have extra complexities (such as parsing and rendering) which, whilst they require careful programming, are not the problems which STAGE has been designed to solve. We defined the following requirements for the game:

- When a user leaves his computer and moves onto a mobile device (potentially to commute to work) he should be able to migrate his game from his fixed computer to his mobile device or vice-versa.
- A computer 'hosting' the game server leaves the network. The game's components must be redistributed about the nodes. It is important that the components of the game (for example user manager, logger, etc.) need not all migrate to the same host. This makes the system more scalable as no one node is responsible for all the extra processing.
- The game should initially be written under the assumption that users are 'local' and it should be adapted to allow those users to exist on remote nodes.
- The game should scale. Players should not notice a significant (ideally no) performance degradation as more players join the game.

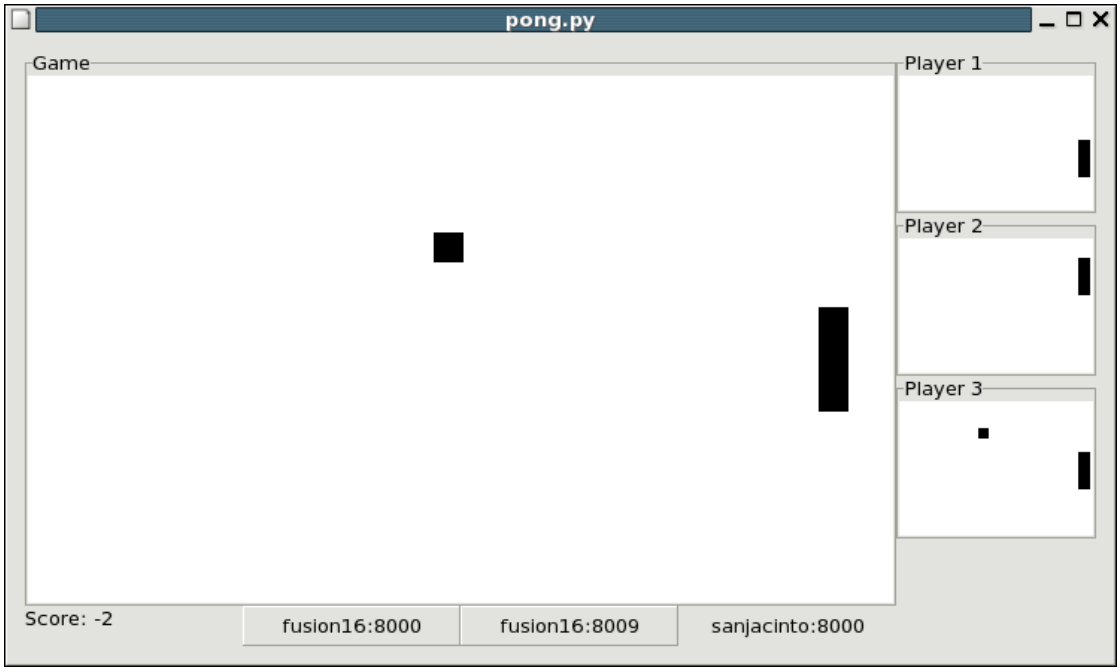

Figure 6.9: Distributed Pong with three players.

Development

The game selected was Pong. The final game is shown in figure 6.9. Pong was the first large application developed in STAGE. The development process was used to shape the language: to determine which features were missing and to determine additional features that would enhance the language. This makes analysis of development time unwise since much of the time spent on development of Pong was time spent enhancing the language. This comprised of adding Driver Actors for GUI elements, developing Actor-orientated drawing tools and keywords for more exotic synchronisation patterns.

6.3.2 Distributed web-framework

Many small dynamic web-applications are hosted on a single host which contains both a web server coupled with a suitable scripting language and possibly a data store. A single host is unsuitable for enterprise quality applications since their requirements and operating environments often include high load, reliability and fault tolerance. To achieve these criteria the components are often distributed across hosts. A typical architecture is shown in figure 6.10, although most real-world infrastructures include further web servers - I have omitted these for clarity.

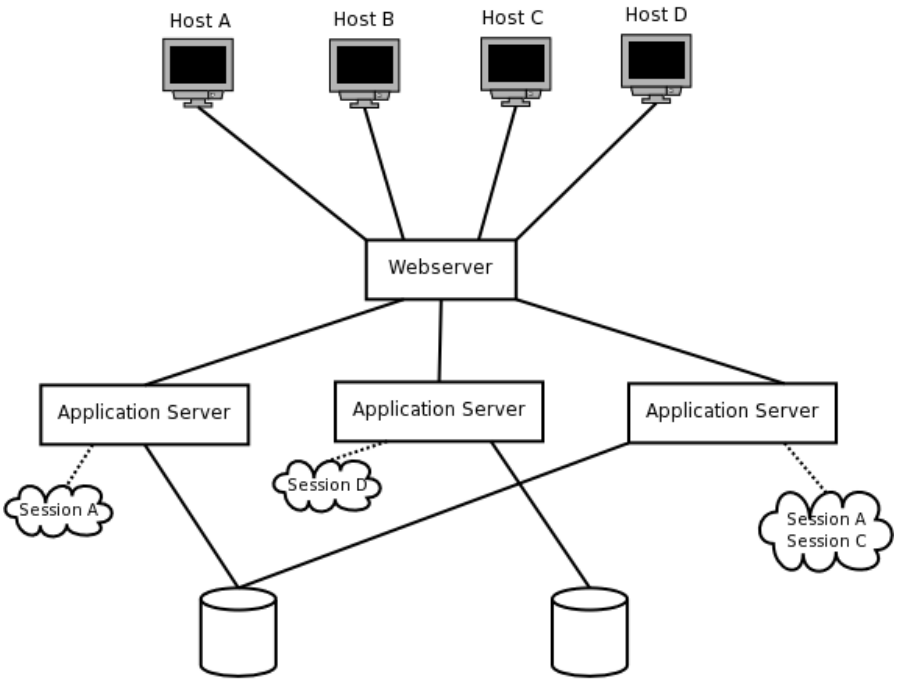

Figure 6.10: A typical web-application infrastructure

Challenges

One problem faced by such systems is state. The HTTP request-response cycle makes it inherently stateless. To provide state most applications store user-specific data in memory-based sessions. The disadvantage of this approach is that, ignoring session replication, a user's request must be served by the application server which is 'responsible' for the user's session. This constrains load balancing and can result in a poorly performing host being forced to service requests because it is the only host that has the relevant session data. Similarly hosts cannot be restarted or shut down until all the sessions they are maintaining have expired.

To increase the capacity of such systems extra hosts are added. However these must be installed, configured and deployed before they can actively service requests. This takes both time and expertise. A host must be assigned a role, either web server, application server or database server. This represents inflexibility - if the web servers are underloaded and the application servers are overloaded then this is a sub-optimal use of resources. It would be beneficial to be able change the role, or provide dual roles for hosts dynamically at runtime.

STAGE's solution

Instead of giving each host a role, each host runs the STAGE execution environment. Actors, not hosts, represent the roles - web server Actors serve static content and application Actors perform processing. The Actors are free to migrate between hosts to take advantage of the

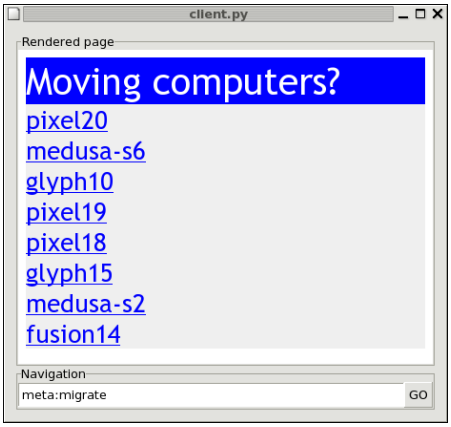

Figure 6.11: Moving hosts using the Actor browser.

computational power spread amongst the network. Actors may migrate from a host to allow it to restart, which allows hosts to undergo maintenance without disruption to the system. Since 'Everything is an Actor' sessions are also free to migrate which means users requests are not tied to a host, which allows more flexibility in load balancing.

To test the system I combined this example with the network monitor of section 6.1.1. The result is a web service that allows a system administrator to monitor hosts from a web browser. The system stores the results of previous status probes in a session variable and displays the saved information each time a user visits the site. Users can request that the host is re-probed and the data updated. The site is shown in figure 6.12. Initially I aimed to support interaction via standard browsers however it soon became apparent that creating an Actor model of a socket, which would provide an external interface for real-world browsers, is non trivial especially if error conditions are to be handled correctly. For this reason a created my own Actor-based browser. Unfortunately this means that the conversation between the browser and the server is not strictly HTTP since the communication is encapsulated in STAGE's transport layer however the communication does follow the request-response pattern of HTTP. Creating my own Actor-based browser confers the benefit that the browser itself can migrate. The migration screen is shown in figure 6.11.

It is with regret that I have not had the time to gather quantitative performance data from such a system. I am able to verify that the system remains operational as hosts enter and exit the system.

6.4 Conclusion

Standard benchmarks for Actor languages are sorely missing. As such each language developer measures the performance of their language using their own set of evaluation tasks. This make inter-language comparison hard - each language developer selects those tasks which show their language in a favourable light. I have tried to avoid this temptation. The Armstrong Challenge is a test that is best suited to languages with fast user-land schedulers. My language uses Oper-

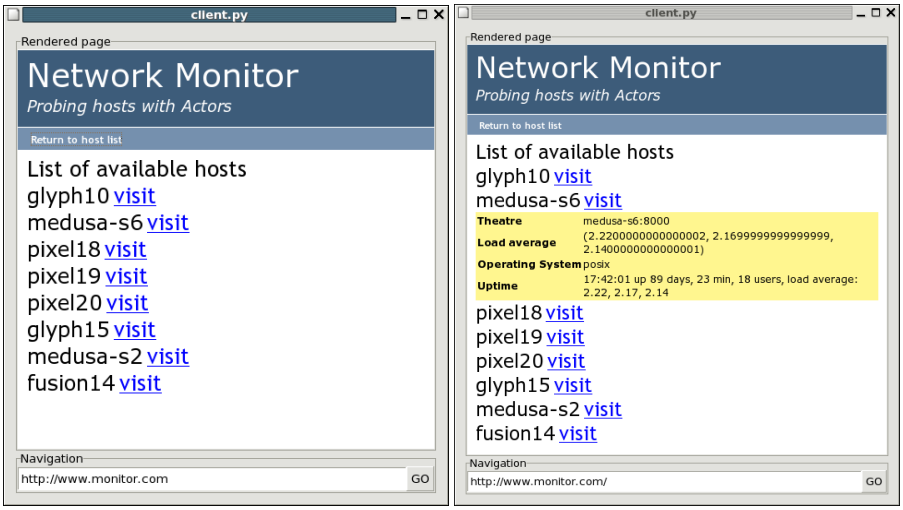

Figure 6.12: Monitoring hosts and storing the results using sessions.

ating System threads but nonetheless I have subjected my language to this test and it performs favourably. Testing the extent to which STAGE is a 'good' language has been difficult. Short examples often hide the complexity inherent in large systems and favour more lightweight languages. The best way to test a language is to use it a lot and to build large systems.

Chapter 7

Conclusions & further work

This section outlines the insights that I have gained from both implementing an Actor-based language and using the completed language.

7.1 The language

> "Writing anything in glint is disastrously hard as you really have to do manual hackery to get basic numbers to work"
>
> Matthew Sackman on his language GLINT.

I believe my decision to incorporate STAGE into PYTHON was a correct one. Many languages start from a 'blank sheet' and define the language from grammar through to runtime. The danger with this approach is that implementers are forced to provide features which programmers have come to expect from modern languages. These features include scoping rules, operators, garbage collection, types, object/data declarations, inheritance and initialisation sugar for common datatypes (this list is by no means exhaustive). These are not related, at least directly, to the Actor model and are mainly un-novel. Many Actor language implementers forgo at least some of the features outlined due to time constraints or lack of interest. This leaves some Actor languages with immature, unintuitive or confusing functionality rendering them unsuitable for real-world development. By using PYTHON I have not had to expend time building these well documented run-of-the-mill features. I have been free to experiment with and implement interesting features. By concentrating on the Actor-specific features I have not lost any of features that programmers expect and rely on.

I made use of the ECLIPSE IDE combined with the PyDEV[1] ECLIPSE plug-in to develop STAGE applications. This combination provides syntax highlighting, code completion, error checking and code navigation for PYTHON. Since STAGE does not subvert PYTHON significantly, ECLIPSE and the plug-in were able to provide the same tool support for STAGE code. When researching

[1] PyDEV ECLIPSE plug-in http://pydev.sourceforge.net/

SALSA I wrote a number of exploratory examples and although SALSA Actors' bodies are defined in JAVA, ECLIPSE was unable to offer language support because Actor declarations differ syntactically from JAVA's class definitions. This lack of support highlighted the benefits of tool support for a language.

7.2 The Actor model

I have explored a number of Actor-based languages during the research phase of this project. I have found that the Actor model's greatest strength is that code does not require user-visible locking. Care still needs to be taken to prevent invalidation of higher-level integrity constraints but the lack of shared mutable state prevents a number of common errors.

Most Actor languages embellish or the Actor model to some extent. It is clear that Actor languages which do not provide higher-level abstractions such as call-and-return and error semantics are harder to use. In some sense the Actor model is an excellent assembly language or basis for a more usable concurrent language. Of the languages surveyed only two use static typing: SALSA and SCALA. STAGE inherits PYTHON's dynamic typing, however I am unsure as to whether static typing is of significant benefit to Actor-languages. Static typing certainly helps alleviate some compile-time errors but restricts flexibility. STAGE Actors need to know nothing about an Actor they wish to interact with other than its name. Actors can send a message with a given name (using the method call abstraction). The target Actor will either accept it or will reject it and raise an error. This is closer to a capability model which I favour.

7.3 The execution environment

The STAGE execution environment is a large and complex threaded application written in PYTHON. Throughout its construction I utilised PYTHON's threading module for synchronisation. PYTHON's threading model borrows heavily from JAVA and the primary mechanism for thread control is by forming monitors. I found lock based synchronisation both tedious and error prone. During development I discovered for myself the difficulties associated with testing concurrent interaction at the unit-test level - errors would occur, but I was often unable to reproduce them. Only careful reading of the code would reveal the source of the error. I noted also that error handling across threads is non-trivial. Which thread should handle the error? How is the error communicated? Could communicating the error cause a further error? These questions do not need answering in a single threaded contexts where errors are simply thrown upwards, unwinding the call stack until a component is willing to handle it.

My intention was to use the LTSA tool to model key components however I found that my inexperience with the tool prohibited me from modelling the complex locking found in this project.

7.4 Further work

The section outlines the areas of STAGE that are suitable for further work and investigation.

STAGE does not distinguish between local and remote communication. The realtime game demonstrated that the system becomes unusable if two Actors which engage in high-volume realtime message passing are located on different hosts. To prevent such situations from arising Actors can specify 'friends' which are Actors with which they have particular attachment and should not become separated from. This is discussed in more detail in section 5.3.8 and the existing execution environment keeps 'friends' together. However this requires programmers to explicitly specify an Actor's friends. Further work should be undertaken to automatically identify 'friends' based on the volume and size of messages sent between them.

The PYTHON global interpreter lock presented a problem late in development. A host with multiple cores would not experience the expected speedup. This turned out to be due to the global interpreter lock which prevents two PYTHON bytecodes from being executed at the same time. As this was not noticed until late in development I did not have a chance to provide a well engineered solution, rather I ensured that hosts with multiple cores created multiple instances of the STAGE execution environment bound to distinct ports - one for each core. Unfortunately this means that Actors must essentially migrate between cores. This can be automated by using the load balancer, however a more rigorous solution would enable a single theatre to spawn multiple PYTHON interpreters to allow seamless multiple-core execution.

STAGE's naming strategy is outlined in section 5.4.2. In brief locating Actors is based on 'hints' whereby location mechanisms tell an Actor the last known location of a target Actor. The current implementation only includes one source of hints. Providing compatibility with existing naming mechanisms such as DNS and flat files would be beneficial. ActorSpaces [9] allow advanced naming. For example as a client Actor we may be interested in retrieving the contents of a file. Assuming there are three FileServers (FileServer1, FileServer2 and FileServer3) then we must select one to service out request. Advanced naming allows us to make a request to "FileServer*" and we will be connected with one of the FileServers. This would be a useful addition to STAGE.

STAGE is most suited to trusted environments - it does not attempt to deal with security, authenticity or corruption. Ideally theatres would each maintain a public-private key pair. Each Actor can then identify hosts by their public key and can collaborate with other Actors to determine if it should trust a host before communicating with or migrating to it. Theatres are at risk from malicious Actors - theatres trust that incoming Actors are courteous and will subject themselves to load balancing. Similarly theatres assume Actors will not try to damage the host. These assumptions are valid for a trusted LAN but are unjustifiable in a hostile environment like the Internet. In these environments Actors' scripts should be signed by the author or originating host to assert that they are 'well-behaved' Actors.

Since I am the only developer of STAGE applications I have not produced the copious documentation that accompanies most production languages. If STAGE has mass appeal then documentation and tutorials will be required to introduce programmers to STAGE.

STAGE messages are PYTHON specific. Data types in messages are communicated as serialised PYTHON objects. If we were to replace this with a more abstract communication format such

as XML, then code written in other (non-PYTHON) languages would be able to communicate with STAGE Actors. This interoperability would benefit STAGE since tasks which require high-performance computation could be written in a lower-level language, such as C, and the results communicated back to the relevant Actor when completed.

7.5 Concluding remarks

The STAGE language inherits much of its flexibility from its implementation language, PYTHON. STAGE programs can be developed quickly without the need for extra frameworks for concurrency, distribution or naming. For this reason I feel STAGE would be an excellent language for quickly prototyping or modelling distributed systems before beginning expensive development in a less dynamic/abstract, more rigid language such as JAVA. I feel that STAGE's mobility constructs are simple and intuitive and would make it useful as an assembly language for agents or to demonstrate mobility.

Appendix A

Numerical approximation results

A.1 Trapezoidal approximation on one host

This section outlines the methodology and results gathered when running the trapezoidal approximation benchmark. Each experiment was run on the same host, `flathead.doc.ic.ac.uk`. The specifications of this host are given below. I am indebted to Ian Ballantyne for allowing me to run these disruptive tests on his project machine.

Hostname	flathead.doc.ic.ac.uk
Operating System	Linux 2.6.17.11-x86_01doc #1 SMP
Processor	i686 Intel(R) Core(TM)2 CPU 6400 @ 2.13GHz

A.1.1 Python

The PYTHON (single threaded) implementation is given in listing 44. The runtimes follow.

Trapezoids	Computed result	Time/s
1	-6.45161622902	0.000
10	0.280104600801	0.000
100	-0.00582367312068	0.001
1000	-0.600651437975	0.008
10000	-0.602940026915	0.080
50000	-0.603369078095	0.417
100000	-0.603381402428	0.808
250000	-0.6033848991	2.029
500000	-0.603385412246	3.931
750000	-0.603385509851	5.903
1000000	-0.603385544705	8.009

Listing 44 Trapezoidal approximation - PYTHON implementation.

```
1  import time
2  import math
3  import sys
4
5  def main(trapezoids):
6    start = time.time()
7    result = approximate(trapezoids, 0, 20)
8    end = time.time()
9    print "Result: %s Runtime: %s" % (result, end - start)
10
11 def approximate(n, lower, upper):
12     result = 0.0
13     h = (upper - lower) / float(n)
14     for i in range(1, n+1):
15       result += (f(xi(i, h)) + f(xi(i+1, h))) * (h / 2)
16     return result
17
18 def f(x):
19  return (math.sin(x**3.0-1.0)/(x+1))*math.sqrt(1.0+math.exp(math.sqrt(2.0*x)))
20
21 def xi(i, h):
22   return (i-1) * h
23
24 if __name__ == '__main__':
25   main(int(sys.argv[1]))
```

A.1.2 Stage

To evaluate the STAGE's performance I created two implementations. One which used one Actor and was similar to the PYTHON implementation. In the second implementation I used two Actors. Each Actor performed an equal share of the work. The runtimes are given below.

Stage (one Actor)

Trapezoids	Computed result	Time/s
1	-6.45161622902	0.003
10	0.280104600801	0.003
100	-0.00582367312068	0.004
1000	-0.600651437975	0.011
10000	-0.602940026915	0.086
50000	-0.603369078095	0.446
100000	-0.603381402429	0.912
250000	-0.6033848991	2.295
500000	-0.603385412246	4.103
750000	-0.603385509851	6.694
1000000	-0.603385544705	8.273

Stage (two Actors)

Trapezoids	Computed result	Time/s
1	0.000000000000	0.510
10	0.280104600801	0.513
100	-0.00582367312068	0.515
1000	-0.600651437975	0.514
10000	-0.602940026915	0.510
50000	-0.603369078095	0.516
100000	-0.603381402429	0.612
250000	-0.6033848991	1.331
500000	-0.603385412246	2.559
750000	-0.603385509851	3.318
1000000	-0.603385544705	4.237

A.1.3 Java

The JAVA implementation is given in listing 45. Since SALSA rewrites into JAVA this provides a suitable comparison. The runtime of the JAVA implementation is given below:

Listing 45 Trapezoidal approximation - JAVA implementation.

```
public class Trap {

  void start(int trapezoids) {
    long start = System.currentTimeMillis();
    double result = approximate(trapezoids, 0, 20);
    long end = System.currentTimeMillis();
    System.out.printf("Result: %s Runtime: %s\n", result, (end - start) / 1000.0);
  }

  double approximate(int n, int lower, int upper) {
    double result = 0.0;
    double h = (upper - lower) / ((float) n);
    for (int i = 1; i <= n; i++) {
      result += (f(xi(i, h)) + f(xi(i + 1, h))) * (h / 2);
    }
    return result;
  }

  double f(double x) {
    return (Math.sin(Math.pow(x, 3.0) - 1.0) / (x + 1.0))
           * Math.sqrt(1.0 + Math.exp(Math.sqrt(2.0 * x)));
  }

  double xi(int i, double h) {
    return (i - 1) * h;
  }

  public static void main(String[] args) {
     new Trap().start(Integer.parseInt(args[0]));
  }
}
```

Trapezoids	Computed result	Time/s
10	0.2801046008008936	0.000
100	-0.0058807611613284905	0.001
1000	-0.6005913423215166	0.003
10000	-0.6029397850486279	0.031
50000	-0.603369347929124	0.075
100000	-0.6033816762416225	0.125

A.1.4 Salsa

SALSA is quite an esoteric language and one which I do not have much experience with. For this reason I used the implementation given by the SALSA developers [1]. I have modified the timing points to ensure they are consistent with the timing points in the other implementations. The runtimes are given below:

Salsa (one Actor)

Trapezoids	Computed result	Time/s
10	-2.528862835498781	0.368
100	-0.4667658965970356	0.538
1000	-0.6468640606205528	1.873
10000	-0.6031853172111705	4.837
50000	-0.603856395545661	30.059
100000	-0.6036784098289072	77.153

Salsa (two Actors)

Trapezoids	Result	Time/s
10	-4.035210837051882	0.312
100	-0.34260503649302004	0.331
1000	-0.6518571849948178	0.988
10000	-0.602227741421772	4.409
50000	-0.6038155147372892	22.633
100000	-0.6036681628001809	59.178

A.2 Trapezoidal Approximation with distribution

To test STAGE's performance under distribution I used a small set of homogeneous hosts connected via the department's LAN. The specifications are given below:

Hostnames pixel{i}.doc.ic.ac.uk
Operating System Linux 2.6.17.11-x86_01doc #1 SMP
Processor i686 Intel(R) Pentium(R) 4 CPU @ 3.20GHz

where $i \in \{16, 17, 18, 19, 20, 21\}$.

I varied the number of hosts that were able to join the computation. The runtimes for 1, 2, 4 and 6 hosts are given below:

Stage (6 hosts)

Trapezoids	Computed result	Time/s
100	-0.456660400185	0.065
1000	-0.60744668877	0.066
50000	-0.603390425747	0.207
100000	-0.603403063421	0.278
500000	-0.603425145391	1.072
1000000	-0.603425283572	2.019
5000000	-0.603389910587	9.321
10000000	-0.603389912227	35.366

Stage (4 hosts)

Trapezoids	Computed result	Time/s
100	-0.00582367312068	0.044
1000	-0.600651437975	0.0446
50000	-0.603369078095	0.579
100000	-0.603381402429	0.633
500000	-0.603385412246	3.397
1000000	-0.603385544705	6.133
5000000	-0.603385589636	32.753
10000000	-0.60338559127	49.945

Stage (2 hosts)

Trapezoids	Computed result	Time/s
100	-0.00582367312068	0.528
1000	-0.600651437975	0.531
5000	-0.600390775558	0.526
10000	-0.602940026915	0.528
50000	-0.603369078095	0.642
100000	-0.603381402429	0.961
500000	-0.603385412246	2.878
1000000	-0.603385544705	5.675
5000000	-0.603385589636	28.176
10000000	-0.60338559127	84.387

Stage (1 host)

Trapezoids	Computed result	Time/s
100	-0.00582367312068	0.017
1000	-0.600651437975	0.034
5000	-0.600390775558	0.087
10000	-0.602940026915	0.185
50000	-0.603369078095	0.778
100000	-0.603381402428	1.950
500000	-0.603385412246	11.463
1000000	-0.603385544705	23.904
5000000	-0.603385589636	111.424
10000000	-0.60338559127	108.368

Appendix B

Armstrong challenge results

B.1 Armstrong challenge data

To implement the Armstrong challenge I created a ring of N Actors. Each Actor was supplied a reference to the Actor immediately next (clockwise) in the ring. An Actor was selected as the start node. The start node begins the challenge by sending a token (via method call) to its immediate neighbour. This continues around the ring until the token has been passed 100000 times. Timing starts from before the creation of any ring Actors and ends as soon as the token has been passed a sufficient number of times (100000 times for this benchmark). The per-message time is then calculated from the overall time. I used the host `flathead.doc.ic.ac.uk`, whose specification is given in appendix A.1, to perform the tests on. The results are given below:

Number of Actors	Time per message send/s
10	0.000634981322289
100	0.000587149710655
1000	0.000603391430378
2500	0.000630899238586
5000	0.000681661748886
7500	0.000731625957978
10000	0.000796699509621
12500	0.000866200470924
14285	0.000937933628306

Appendix C

Load balancer results

This test evaluated the performance of the load balancer. I selected four of the pixel{i}.doc.ic.ac.uk hosts. The specifications of these hosts can be found in appendix A.2. The test began with no Actors on any of the hosts except a load balancer Actor on host 1. I then began to add Actors onto host 1, adding one Actor every 200 milliseconds. Each Actor added varied in its requirements - Actors ranged from CPU intensive to relative inactivity. The performance characteristics of each Actor were chosen randomly. The load averages of each of the hosts at 4 second intervals are given below:

Time/s	host 1	host 2	host 3	host4	Time/s	host 1	host 2	host 3	host4
0	0.49	0.11	0.3	0.08	138	8.76	0.48	0.31	0.46
4	0.61	0.1	0.27	0.08	142	8.14	0.48	0.29	0.42
8	0.61	0.09	0.25	0.07	146	8.14	0.52	0.27	0.39
12	0.64	0.09	0.23	0.07	150	7.57	0.56	0.27	0.52
16	0.67	0.08	0.21	0.06	154	7.04	0.59	0.24	0.52
20	0.7	0.08	0.21	0.06	158	6.56	0.55	0.22	0.55
24	0.72	0.07	0.19	0.06	162	6.11	0.55	0.21	0.51
28	0.72	0.06	0.18	0.13	166	6.11	0.5	0.19	0.55
32	0.74	0.06	0.16	0.12	170	5.7	0.46	0.19	0.5
36	0.76	0.06	0.23	0.19	174	5.33	0.42	0.17	0.5
40	0.78	0.05	0.23	0.18	178	4.98	0.39	0.16	0.54
44	0.88	0.05	0.21	0.18	182	4.66	0.39	0.23	0.58
48	0.97	0.05	0.2	0.24	186	4.53	0.36	0.21	0.53
52	0.97	0.05	0.18	0.22	190	4.53	0.33	0.21	0.57
56	0.97	0.04	0.16	0.28	194	4.24	0.3	0.19	0.57
60	0.97	0.04	0.16	0.26	198	3.98	0.28	0.18	0.6
64	0.97	0.04	0.15	0.26	202	3.75	0.28	0.16	0.64
68	0.98	0.03	0.14	0.32	206	3.52	0.98	0.15	0.66
72	0.98	0.03	0.13	0.29	210	3.52	1.06	0.15	0.69
76	1.06	0.03	0.12	0.43	214	3.4	1.05	0.22	0.69
80	1.05	0.03	0.12	0.4	218	3.29	1.05	0.2	0.72
84	1.05	0.02	0.11	0.4	222	3.11	1.05	0.26	0.74
88	1.04	0.02	0.1	0.68	226	2.94	1.04	0.4	0.68
92	1.04	0.02	0.17	0.71	230	2.94	1.04	0.4	0.62
96	12.41	0.02	0.16	0.65	234	2.86	1.04	1.01	0.62
100	11.66	0.02	0.16	0.6	238	2.71	1.03	1.01	0.65
104	10.88	0.02	0.22	0.6	242	2.58	1.03	1.01	0.6
108	10.09	0.02	0.21	0.55	246	2.45	1.03	1.01	0.55
112	9.36	0.02	0.27	0.51	250	2.45	0.95	1.01	0.51
116	9.36	0.01	0.33	0.55	254	2.41	0.87	1.01	0.51
120	8.69	0.17	0.33	0.5	258	2.3	0.8	1.0	0.55
124	8.08	0.24	0.3	0.5	262	2.2	0.8	1.0	0.58
128	9.27	0.3	0.28	0.54	266	2.1	0.74	1.0	0.62
132	9.09	0.3	0.25	0.5	270	2.01	0.68	1.0	0.65
136	9.09	0.43	0.31	0.46	273	2.01	0.62	1.08	0.65

Bibliography

[1] Comprehensive [salsa] example: Trapezoidal approximation. http://wcl.cs.rpi.edu/salsa/demos/ComprehensiveExample.htm.

[2] The end of moore's law? http://qubit.plh.af.mil/RelatedArticles/related/Mann01.pdf.

[3] Erlang design principles. http://www.erlang.org/doc/design_principles/des_princ.html.

[4] The madkit project. http://www.madkit.org.

[5] Python documentation: Threading - higher-level threading interface.

[6] Unification of active and passive objects in an object-oriented operating system. In *IWOOOS '95: Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, page 68, Washington, DC, USA, 1995. IEEE Computer Society.

[7] *HP-UX Process Management*, April 1997.

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

[9] Gul Agha and C. J. Callsen. Actorspace: An open distributed programming paradigm. In *Proceedings 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, pages 23–323, 1993.

[10] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[11] Alexander Ahern and Nobuko Yoshida. Formalising java rmi with explicit code mobility. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 403–422, New York, NY, USA, 2005. ACM Press.

[12] J. Armstrong, S. Virding, and M. Williams. Use of Prolog for Developing a New Programming Language. In C. Moss and K. Bowen, editors, *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, 1992. Association for Logic Programming.

[13] Joe Armstrong. The development of erlang. *SIGPLAN Not.*, 32(8):196–203, 1997.

[14] Joe Armstrong. Concurrency oriented programming in erlang. http://ll2.ai.mit.edu/talks/armstrong.pdf, 2002.

[15] Luiz A. Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(02):22–28, March 2003.

[16] A. Bieszczad, T. White, and B. Pagurek. Mobile agents for network management. *IEEE Communications Surveys*, 1998.

[17] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[18] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[19] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and Synchronization in Concurrent OOP. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the ECOOP '87 European Conference on Object-oriented Programming*, pages 32–40, Paris, France, 1987. Springer Verlag.

[20] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Weak and Strong Mobility in Mobile Agent Applications. In *Proceedings of, 2nd International Conference and Exhibition on The Practical Application of Java*, PA JAVA, Manchester, UK, April 2000. The Practical Application Company Ltd.

[21] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[22] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[23] Ben. Crystall. Faster faster. *New Scientist*, 168(2270):60, 2000.

[24] Travis Desell. Personal communication, 2007.

[25] Travis Desell, Kaoutar El Maghraoui, and Carlos Varela. Load balancing of autonomous actors over dynamic networks. *hicss*, 09:90268a, 2004.

[26] Sophia Drossopoulou, Susan Eisenbach, and Alexis Petrounias. A concurrency model of chorded languages. December 2005.

[27] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is java binary compatibility? In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 341–361, New York, NY, USA, 1998. ACM Press.

[28] Lawrence Flon. On research in structured programming. *SIGPLAN Not.*, 10(10):16–17, 1975.

[29] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, Reading/Massachusetts, 1999.

[30] Martin Fowler. Fluent interfaces. http://www.martinfowler.com/bliki/FluentInterface.html, 2005.

[31] Martin Fowler. Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/languageWorkbench.html, 2005.

[32] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[34] B. Garbinato and R. Guerraoui. Using the strategy design pattern to compose reliable distributed protocols. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS-3)*, pages 221–232, Portland, Oregon, USA, 1997.

[35] Gnutella. The gnutella protocol specification v0.4, 2003.

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[37] Philipp Haller. An object-oriented programming model for event-based actors. Master's thesis, Fakultt fr Informatik, Universitt Karlsruhe, 2006.

[38] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, Springer LNCS, 2006.

[39] Philipp Haller and Martin Odersky. Actors that Unify Threads and Events. Technical report, 2007.

[40] Baker Henry G., Jr. and Carl Hewitt. The incremental garbage collection of processes. Technical Report AIM-454, 1977.

[41] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of the 3rd IJCAI*, pages 235–245, Stanford, MA, 1973.

[42] C. Hewitt and Peter de Jong. Open systems. AIM 691, AI Lab, MIT, Cambridge, MA, 1982.

[43] Carl. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.

[44] IEEE. *802.3 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*. IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, Mar 2002.

[45] Dan Kegel. The c10k problem. http://www.kegel.com/c10k.html.

[46] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. Seti@home-massively distributed computing for seti. *IEEE MultiMedia*, 3(1):78–83, 1996.

[47] Don. Kretsch. Beware: Concurrent applications are closer than you think! http://developers.sun.com/sunstudio/articles/overview_concurrent.html.

[48] Ralf Lammel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press.

[49] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[50] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[51] Doug Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

[52] H. Lieberman. A preview of act 1. *MIT AI Memo*, (625), 1981.

[53] H. Lieberman. Thinking about lots of things at once without getting confused: Parallellism in act 1. *MIT AI Memo*, (626), May 1981.

[54] Seth. Lloyd. Ultimate physical limits to computation. *Nature*, 406(6799):1047–1054, 2000.

[55] Ignac Lovrek, Gordan Jezic, Mario Kusek, Igor Ljubi, Antun Caric, Darko Huljenic, Sasa Desic, and Ozren Labor. Improving software maintenance by using agent-based remote maintenance shell. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 440, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[57] J.A McCann. Process and concurrency. http://www.doc.ic.ac.uk/~jamm/teaching/osconcepts/OS4.pdf.

[58] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients, 2002.

[59] Sun Microsystems. New i/o [nio] apis. http://java.sun.com/j2se/1.4.2/docs/guide/nio/, 2002.

[60] Giuseppe Milicia and Vladimiro Sassone. The inheritance anomaly: Ten years after.

[61] Michael O'Connell. Java: The inside story.

[62] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[63] Karen Osmond, Olav Beckmann, and Paul Kelly. A Domain-Specific Interpreter for Parallelizing a Large Mixed-Language Visualisation Application. In *LCPC 2005*, October 2005.

[64] John. Ousterhout. Why threads are a bad idea (for most purposes). http://www.cs.vu.nl/~gpierre/courses/np/ousterhout.pdf.

[65] Paul A. Packan. Device physics:pushing the limits. *Science*, 285(5436):2079–2081, 1999.

[66] Chris Rathman. Shapes oo example: Erlang code. http://www.angelfire.com/tx4/cus/shapes/erlang.html.

[67] Matthew Sackman. Glint: Breeding mobile ambients with actors. Master's thesis, Department of Computing, Imperial College, 80 Queensgate, London SW7 2BZ, 2006.

[68] Matthew Sackman and Susan Eisenbach. Glint: Addressing concurrency through language design. 2006.

[69] Fabrizio Sebastiani. Towards a logical reconstruction of information retrieval theory. *Cybernetics and Systems*, 30(5):411–428, 1999.

[70] R. G. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. pages 357–366, 1988.

[71] Herb. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr Dobb's Journal*, 30(3), 2005.

[72] Anand Tripathi and Robert Miller. Exception handling in agent-oriented systems. pages 128–146, 2001.

[73] Guido van Rossum. The PYTHON tutoial. http://docs.python.org/tut/.

[74] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, 2001.

[75] Bill Venners and Erich Gamma. Design principles from design patterns. a conversation with erich gamma, part iii. http://www.artima.com/lejava/articles/designprinciples.html, 2005.

[76] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994.

[77] Tim Walsh, Paddy Nixon, and Simon Dobson. As strong as possible mobility: an architecture for stateful object migration on the Internet. Technical Report TCD-CS-2000-11, Department of Computer Science, Trinity College Dublin, February 2000.

[78] Wei-Jen Wang and Carlos A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. Technical Report 06-04, Dept. of Computer Science, R.P.I., February 2006. Extended Version of GPC'06 Paper.

[79] N. Williams. Morphable objects in smalltalk. In *ASWEC '98: Proceedings of the Australian Software Engineering Conference*, page 48, Washington, DC, USA, 1998. IEEE Computer Society.

[80] W. A. Wulf. A case against the goto. pages 83–98, 1979.

[81] Steve Yegge. Execution in the kingdom of nouns. http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html.