Stage: Python with Actors

John Ayres
Department of Computing,
Imperial College London
jwa@doc.ic.ac.uk

Susan Eisenbach
Department of Computing,
Imperial College London
S.Eisenbach@imperial.ac.uk

Abstract

Programmers hoping to exploit multi-core processors must split their applications into threads suitable for independent, concurrent execution. The lock-based concurrency of many existing languages is clumsy and error prone – a barrier to writing fast and correct concurrent code.

The Actor model exudes concurrency – each entity in the model (an Actor) executes concurrently. Interaction is restricted to message passing which prevents many of the errors associated with shared mutable state and locking, the common alternative. By favouring message passing over method calling the Actor model makes distribution straightforward.

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 is functional; a paradigm unfamiliar to many programmers. There is a need for a language that presents a familiar and fast encoding of the Actor model. In this paper we present* STAGE*, our mobile Actor language based on* PYTHON.

1 Introduction

The most popular programming languages were designed well before multi-core computers became mainstream and before the widespread use of mobile and pervasive devices. To program many of today's applications, running on these architectures requires concurrency, distribution, parallelism, and/or mobility. Concurrent applications are vulnerable to a range of errors not present in single threaded code. Using existing mechanisms with shared data, protecting data from low level corruption caused by concurrent access is error prone.

To increase performance applications may require additional hosts to provide further scope for parallisation. In some applications distribution is essential to provide access to information that is only available remotely, or to gather information from remote locations.

The advent of mobile and pervasive computing has given rise to a new set of operating conditions. We are carrying an increasing number of mobile devices from laptops to mobile phones and PDAs. These devices are synonymous with low bandwidth communication operating over intermittent connection mediums. 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 neighbours. 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. From a user's perspective mobility allows user-visible applications such as email clients and web browsers to move between hosts. Consider beginning to compose an email on a desktop computer, clicking a button to move it to to a mobile device, and then finishing and sending it during a commute to work.

In the Actor model of Hewitt, Bishop, Steiger [11], and Agha [2] the underlying behaviour is that of independent execution and asynchronous communication. Actors are active entities and only interact by exchanging messages (so no shared state). The model can be extended to allow such messages to span hosts on a network. Actors encapsulate both data and behaviour (the 'script') and can be extended to support distribution and process mobility. Actors can only communicate with acquaintances that they have successfully accrued during their lifetime. In addition to sending messages, Actors can create other Actors and can alter their behaviour.

Existing Actor-based languages often seek out a particular niche. SALSA [16] favours distribution at the cost of local speed, SCALA featuring event-based Actors [10] favours

lightweight switching and supports a high volume of Actors, and ERLANG [3] favours reliability and development speed. Whilst these languages are successful in achieving their aims, they are often too specialised to be of general appeal. We have created the STAGE language to present abstractions that make the Actor model more consistent with Object Orientated methodologies.

The language takes PYTHON, a dynamic Object Orientated language, and extends and modifies the existing language constructs to provide a new Actor language that combines the lightweight syntax of PYTHON with powerful Actor-language abstractions. STAGE is available from [5].

The remainder of this paper is organised as follows: In Section 2 we present example programs to introduce STAGE. The design decisions for the STAGE language including distributed naming and rendezvous, dynamic networks, interaction with the Operating System, load balancing, 'lazy synchronisation', based on Lieberman's futures [13] are presented in 3. The implementation details are in Section 4. We present measurements that demonstrate that the language is capable of increasing performance in multicore and distributed contexts in Section 5. We compare our design with other Actor languages in 6 and in Section 7 we present a selection of features and improvements that could be made to enhance the language.

2 Programming in STAGE

Actor definitions follow PYTHON's class definitions. Defining a method on an Actor represents an Actor's capability to accept, process and possibly respond to a message of a specific type. The code below for a simple chat program contains a definition, instantiation and use of a `User` Actor that exposes two capabilities: 'talk' and 'listen'.

```
1  class ChatServer(BidirectionalServer):
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.listen)
12     Keyboard().subscribe(self.talk)
13
14   def talk(self, msg):
15     self.server.say(msg)
16
17   def listen(self, msg):
18     print ">", msg
19
20 john = User()
21 susan = User()
```

Lines 7 through 18 specify the behaviour or 'script' `User` Actors must follow. Line 9 is the special 'birth' method that is executed when an Actor is instantiated.

Lines 14 through 18 specify the external interface of `User` Actors and the behaviour `User` instances should exhibit upon receipt of `talk` and `listen` messages[1]. The `ChatServer` Actor maintains a reference to all of its clients and its clients know the unique name of the server, a behaviour inherited from `BidirectionalServer`.

Writing a simple chat program in JAVA (without resorting to third-party libraries) would be quite different. The source would be many times longer and likely to have URL and port combinations for locating the server; STAGE's `ChatServer` has a location independent name, which is resolved by the runtime. The STAGE solution would be more reliable than a simple JAVA solution because 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 and the system continues without disruption. The STAGE solution does not contain keywords pertaining to synchronisation or threading, whereas synchronisation and thread creation would be scattered throughout the JAVA code. A JAVA server must be explicitly started up, whereas in STAGE the server is started automatically when a request is made to it. Finally, a JAVA chat program is likely to use persistent connections to transfer the chat text whereas the STAGE Actors only connect to transfer messages and use a retry strategy to deliver messages in the presence of failure.

2.1 Network monitoring

For an example mobile application consider a set of hosts each running the STAGE interpreter where Actors can move freely between hosts to warn of impending problems (low disk space etc) or nefarious activity (network card in promiscuous mode etc.) as shown in Figure 1.

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. We can migrate arbitrary probes or monitors, we may migrate a more computationally 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 [7].

The following code demonstrates the simplicity of migration in STAGE. The `arrived` method on line 8 specifies that upon arrival at a host the Actor should install a virus checker, perform a local probe and then report back the re-

[1] In PYTHON `self` (which refers to an Actor instance) is explicitly passed as the leftmost parameter and access to instance variables and methods must be made explicitly through the `self` variable.

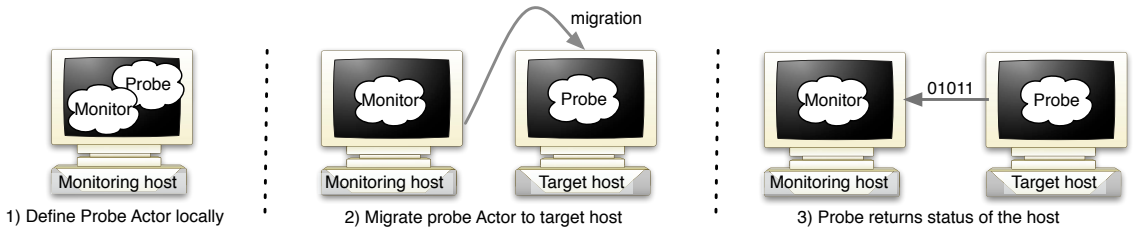


Figure 1. A scheme for monitoring hosts with Actors.

sults. The probe is deliberately over simplistic – probes may be of arbitrary complexity.

```
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      install_virus_checker()
10     data = self.do_probe()
11     response = summarise(data)
12     self.report(response)
13
14   def do_probe(self):
15     status = TheatreStatus()
16     return {'Theatre'          : self.dest,
17             'Load average'     : status.load(),
18             'Operating System' : status.osname(),
19             'Uptime'           : status.uptime()}
```

In a language without migration support 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 difficulties are encountered when we wish to update the behaviour of the probe, or remove the probe completely – a problem STAGE overcomes at the language level through mobility.

3 The STAGE Language

Actor-Actor interaction is, by default, asynchronous. Synchronous result capture (waiting for an acquaintance to return a result) is provided by the `sync` keyword, which can be applied at any point. In the following code sample line one is an example of call-site synchronisation and line four demonstrates that synchronisation can be applied right up to the point at which a result is required:

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

The computed value of `money` is not required until it is printed. Up to this point the variable `money` contains a partial or 'marker' result, a construct inspired by 'futures' [13], which are a receipt for work not yet completed.

Polling interactions occur frequently in applications where a request-response interaction is employed and are supported by the `ready` keyword, which takes a partially computed result and determines if the computed result is ready yet. The following code sample demonstrates a use of `ready`:

```
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."
```

A handler method is 'installed' using the `handle` primitive and automatically executed upon receipt of a suitable message. It is inspired by SALSA's token passing continuations [16] and SCALA's event based Actors [14, 10]. A client can interact with the bank using a handler:

```
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
```

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, and the use of fluent interfaces [9] in STAGE generate such chains. Lazy synchronisation is provided to defer waiting for a result until the result is required. Line 1 is without lazy synchronisation and line 2 is with.

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

Consider a `customer` Actor who adds items to a `shopping_cart` Actor before requesting the total cost.

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

Line 1 results in three 'add' messages being sent to the `cart` Actor. The order (in the absence of failures) is guar-

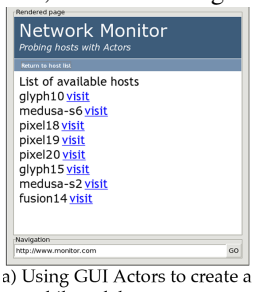
anteed to be the order in which they are sent. The final 'total' message on line 2 *cannot* be received before all the 'add' messages have been received. If an Actor is simultaneously receiving messages from multiple Actors then there will be a non-deterministic interleaving of the messages preserving their independent orderings.

Methods in STAGE can be passed freely between Actors, stored in instance and local variables and are preserved under migration, but cannot be generated on the fly. Actors can even pass methods that form part of other Actors. The snippet below shows how a `WebBrowser` Actor tells a `DNS` Actor to resolve a `url` and forward the resulting IP address to a `PageFetcher` Actor.

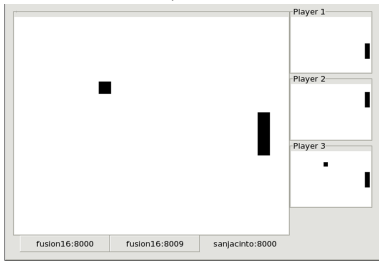
```
1 class WebBrowser(MobileActor):
2
3   def request(self, url):
4     ...
5     dns_server.resolve(page_fetcher.fetch)
6     ...
```

Callbacks can be generalised into a publish/subscribe mechanism where Actors subscribe to an event type by sending a topic name and method pair to the publish/subscribe server. When an event occurs that matches the registered topic the method is invoked. A publish/subscribe mechanism is included in the core library.

To be of general use languages must provide mechanisms for their resultant programs to communicate with the Operating System. The Operating System's capabilities are presented to user mode applications as system calls, which are not consistent with the Actor model. To allow STAGE Actors to interact with the real world we provide a set of *Driver Actors* that provide Actor-friendly representations of underlying system's operations. STAGE provides Driver Actors for common operations but implementers are free to extend this collection, just as JAVA programmers are able to include their own native methods. Two GUI Actors (which, like all Actors, are free to migrate between hosts) are shown below.



a) Using GUI Actors to create a mobile web browser.



b) Distributed Pong with a migrating ball.

STAGE's distribution semantics make no distinction between local and remote Actor-Actor interaction and supports migration through the `migrate_to` primitive. The Actor body executes until it reaches a `migrate_to` statement. The state of the Actor is then frozen and migration is attempted. If successful the Actor resumes its execution in the new theatre. STAGE provides weak mobility: after migration execution continues from code placed in the `arrived` method. If a migration attempt is unsuccessful the `migrate_to` keyword returns an error code and the Actor's execution continues on the original host.

A message that spans hosts (a remote message) is around four to five orders of magnitude slower than a local interaction [17], so it is desirable that groups of Actors that engage in high-volume communication be located in the same theatre. We refer to such groups as *Actor cliques* and a STAGE Actor can specify its friends (other members of the clique). The 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.

An Actor's life-cycle usually follows three distinct phases: (1) birth and initialisation, (2) useful work and interaction and finally (3) death. Rather than include an automatic distributed garbage collector, this version of STAGE provides the `die` primitive for Actor termination.

STAGE provides a set of libraries that can be used to implement load balancing and a `ProbingLoadBalancer` as an example implementation. It consists of the following components:

- An interface Actor that controls registration of worker Actors (those Actors which have expressed a willingness to be load balanced).
- Strategy (or 'decision') Actors that 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 that report back a measure of the load on a host.

The implementation bundled with STAGE uses a naive decision procedure based on theatre load that moves Actors from 'overloaded' to 'underloaded' hosts. Work is in progress to improve STAGE's decision procedure using *Actor satisfaction* [8], which considers queue processing rates.

4 Implementation

STAGE runtime systems have two distinct components. Firstly, there is the theatre, the STAGE execution environment. 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 Fig. 2. A theatre provides support to Actors executing within it. This internal support facilitates Actor behaviour including creating

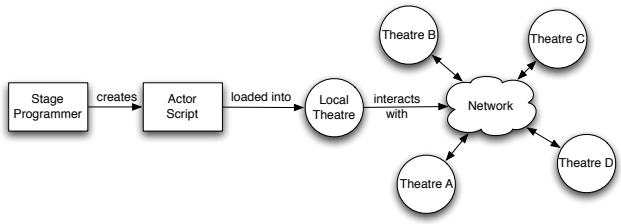


Figure 2. A typical STAGE workflow.

Actors, sending messages and inter-host migration. The execution environment (shown in Fig. 3) 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. We use PYTHON's late binding mechanism to implement Actors. In general, to call a method a caller requests the method *by name* from the object. The object responds with a `callable` object (a functor that 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, modifying 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 to modify object creation. We use this feature to enforce Actor instantiation rules. Migration is achieved through a combination of runtime reflection and dynamic class loading.

The runtime, the communication libraries, migration engine and location services are all implemented in PYTHON. The language maintains most of PYTHON's constructs and features. Threading and external interactions have been replaced by STAGE capabilities. In STAGE methods can be passed and stored in the same way their PYTHON counterparts can, but must be part of the external interface of an Actor.

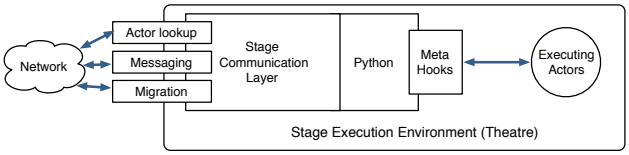


Figure 3. The Stage execution environment.

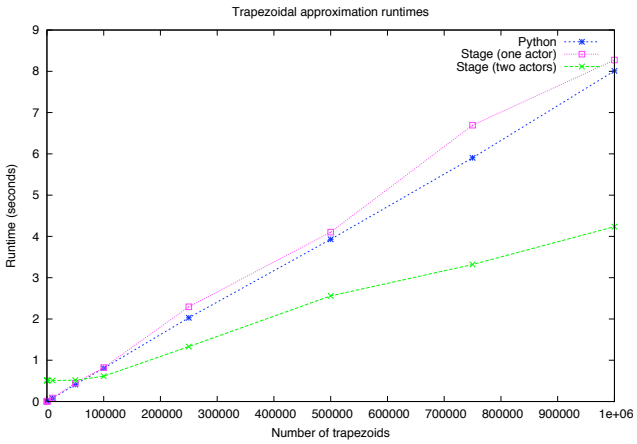


Figure 4. Trapezoidal approximation - runtimes on a multi-core CPU.

5 Performance

5.1 Trapezoidal approximation

STAGE's design focuses on creating a language that promotes fast development of complex distributed systems whist providing increased performance though parallelism and distribution. We chose to test its speed with a trapezoidal approximation of a definite integral as this is a problem that can be easily parallelised. The data for all the runs of the algorithm are in [6]. We used the non-trivial function[2]:

$$f(x) = (\sin(x^3 - 1)/(x + 1)) \times \sqrt{1 + e^{\sqrt{(2x)}}}$$

The algorithm was implemented in STAGE, PYTHON, JAVA, and SALSA.

Fig. 4 shows STAGE's performance using one and two Actors versus a naive PYTHON implementation[3]. We note that STAGE with one Actor is not significantly slower than the PYTHON implementation for larger workloads. STAGE with two Actors exhibits an initial startup cost greater than that of PYTHON. STAGE with two Actors is around twice as fast as PYTHON and STAGE with one Actor. Pleasingly, STAGE's overheads have not adversely affected the performance of PYTHON, and using multiple Actors STAGE achieves a significant speedup.

SALSA is an Actor language that is translated into JAVA and eventually run on a standard JAVA Virtual Machine. Figure 5 shows SALSA's behaviour under load contrasted with that of a naive JAVA implementation and the STAGE

[2]The choice of function was arbitrary, we just wanted something with a significant workload for each Actor.

[3]Running on an i686 Intel Core 2 CPU 2.13GHz and a 2.6 Linux Kernel

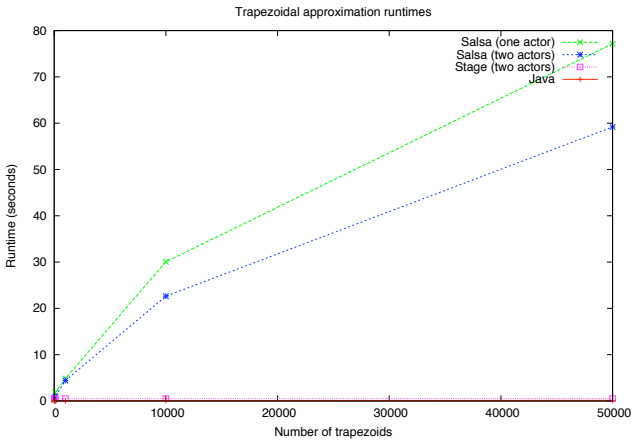


Figure 5. Trapezoidal approximation runtimes - Salsa versus Java and Stage (hugging the x-axis).

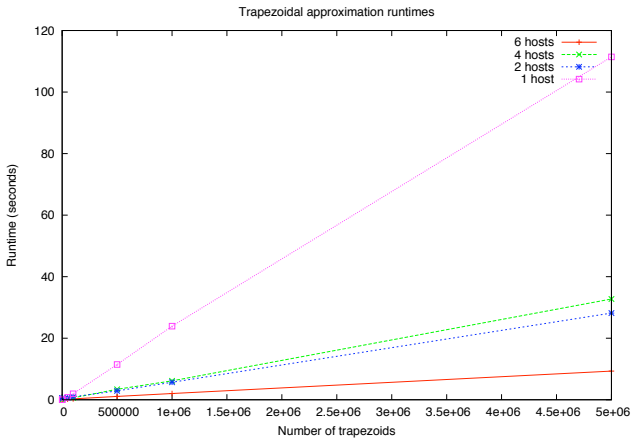


Figure 6. Trapezoidal approximation runtime on multiple hosts using Stage.

(two Actor) implementation. We found that the SALSA program is significantly slower than our JAVA, PYTHON and STAGE implementation and that the two Actor SALSA implementation is not significantly faster than the one Actor implementation.

To gain greater speedups we enlist the help of other hosts. Figure 6 shows how adding more work-willing hosts to the network reduces the runtime. The benchmark is completed 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.

5.2 The Armstrong challenge

ERLANG's designer, Joe Armstrong set out his challenge [4]:

- 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?

Although crafted to highlight the benefits of ERLANG's scheduler rather than representing real work, it is useful 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.

In our solution to the challenge a ring of N nodes (each node represented by an Actor) is created with one node designated as the 'start node'. The start node is presented with a token that it passes around the ring. Once the token has circulated 100000 times the system halts. The network of Actors is timed from the creation of the first Actor to the final halting state. The per message time is calculated by dividing the total running time by the number of messages sent. The results are shown in Fig. 7 and the data can be found in [6].

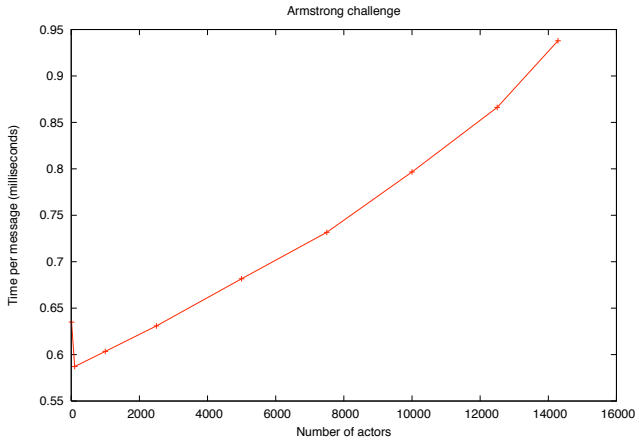


Figure 7. Stage's attempt at the Armstrong challenge.

The results demonstrate the per-message time increases as the number of Actors 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 that the overhead is not crippling - adding 14,000 Actors to a theatre (far beyond the requirements of a reasonable application) only increases the time per message exchange by approximately 0.3 milliseconds.

6 Related Work

The most mature (over 20 years old) of the current Actor languages is ERLANG, a fully featured Actor language which the authors describe as 'a strange mixture, with declarative features (inherited from PROLOG), multitasking and concurrency (inherited from EriPascal and Ada)'. ERLANG provides a solid resilience mechanism based on supervision trees, which reflects its original use in telephony systems. ERLANG's impressive user-space threading model supports fast switching and creation of processes that gives excellent results for the previously mentioned Armstrong challenge. Recent releases of the ERLANG virtual machine create a process per CPU to increase performance on multi-core architectures. STAGE programs run slower than their ERLANG equivalents but we argue that PYTHON-style syntax is more familiar to mainstream programmers.

SALSA [16] (Simple Actor Language System and Architecture) is a implemented as a pre-processor for JAVA [4]. This is both a strength and a weakness. A programmer has access to JAVA's rich set of libraries allowing SALSA Actors to solve 'real world' tasks. However the language does not warn nor prevent inappropriate interaction with JAVA and a list of JAVA libraries and code that will interfere with 'correctness' of a SALSA program is not provided. STAGE does not warn of inappropriate interaction with existing PYTHON libraries but provides *Driver Actors* that offer Actor-friendly abstractions for the operations provided by the underlying Operating System.

SALSA programs 'look' like JAVA programs but the concurrent programming paradigms (i.e. join continuations and token passing) may appear quite alien to mainstream JAVA programmers. JAVA's verbose syntax also hampers SALSA. In the trapezoidal approximation example [1] the logic concerning the Actors' behaviours with respect to concurrent interactions is obscured by the quantity of JAVA code.

SCALA [14] is an Object Orientated, component orientated, functional programming language capable of running on both the JVM and the .Net platform, which aims to fuse Object Orientated and functional programming paradigms. Actors are provided as an additional library [10]. When waiting for a message an (event based) SCALA 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 receives a message it is given a thread from a thread pool so it can continue its execution. This model is advantageous because fewer (JVM/OS) threads are required. 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.

[4]So both compile and runtime errors are JAVA based.

Sillito's language, STAGE [15] (RUBYSTAGE to disambiguate), attempts to combine the concurrency support of ERLANG with RUBY. Like STAGE, RUBYSTAGE is implemented in-language and offers greater support for asynchronous interaction and concurrency than its host language, RUBY. RUBYSTAGE supports ERLANG-style linked processes but does not support inter-host communication or migration. A RUBYSTAGE Actor receives a message through a call to `receive`, which allows for predicate dispatch. We believe that our language presents a language that is both clearer and closer to its host language, PYTHON.

7 Future Work and Conclusions

When working on a new language the temptation is to start with a 'blank sheet' and define the language from grammar through to runtime, but this requires the provision of features which programmers have come to expect. By using PYTHON we have been free to experiment with and implement interesting features without losing any of the features that programmers rely on.

We made use of the ECLIPSE IDE combined with the PYDEV[5] 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 are able to provide the same tool support for STAGE code. Actor languages that provide this rich tool support are in the minority.

Probably the Actor model's greatest strength is that code does not require user-visible locking. Care still needs to be taken to ensure that higher-level integrity constraints are satisfied, but the lack of shared mutable state prevents a number of common errors. As set out by the Actor model, all communication is through the exchange of messages which, by default, are asynchronous. Acquaintances (analogous to collaborators from Object Orientated programming) can be extracted from received messages. All messages are free to span hosts to allow remote communication. To minimise network traffic STAGE, like many modern Actor languages, adds weak mobility to the Actor model.

The Actor model states that Actors can create other Actors and that Actors are able to change their behaviour upon receipt of a message. In STAGE Actor creation follows the usual PYTHON object instantiation call but STAGE protects the resultant Actor from direct access to its internal variables. Upon creation of an Actor the parent Actor learns the name of its child and the child becomes an acquaintance of its parent. Unlike many functional Actor languages STAGE does not provide the ability for an Actor explicitly change its script (commonly via the *become* keyword) rather, like

[5]PYDEV ECLIPSE plug-in `http://pydev.sourceforge.net/`

SALSA et al., Actors specify a replacement behaviour by modifying their internal state.

STAGE does not distinguish between local and remote communication. STAGE networks become choked if two Actors that engage in high-volume realtime message exchange 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 [6] and the existing execution environment keeps 'friends' together. Further work should be undertaken to automatically identify 'friends' based on the volume and size of messages sent between them.

PYTHON's global interpreter lock, which prevents two PYTHON bytecodes from being executed at the same time, hampers multi-core speedups. To overcome this limitation multi-core hosts must create multiple instances of the STAGE execution environment bound to distinct ports or interfaces - one for each core. Unfortunately this means that Actors must migrate between cores, which can be automated by enabling the load balancer. However a better solution would enable a single theatre to spawn multiple PYTHON interpreters to allow seamless multi-core execution.

Locating Actors is based on 'hints' that tell a searching 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.

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 is then able to reliably identify hosts and can collaborate with other Actors to determine if it should trust a host before communicating with or migrating to it. Theatres trust that incoming Actors are harmless and will subject themselves to load balancing. These assumptions 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.

The STAGE language inherits much of its flexibility from PYTHON. STAGE programs can be developed quickly without the need for extra frameworks for concurrency, distribution and naming. By promoting concurrent execution and restricting all inter-Actor communication to message passing STAGE Actors can be distributed both across the cores of a single host and across the network; gaining the performance increases that both entail.

Acknowledgements We are indebted to Matthew Sackman, Neil Dunn and James Dicken for the many productive discussions we had while developing STAGE and Jacob Lee the author of the Python Actor Runtime Library, PARLEY [12], which was developed at the same time as STAGE.

References

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

[2] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] J. Armstrong. The development of Erlang. *SIGPLAN Not.*, 32(8):196–203, 1997.

[4] J. Armstrong. Concurrency Oriented Programming in Erlang. `http://ll2.ai.mit.edu/talks/armstrong.pdf`, 2002.

[5] J. Ayres. Stage homepage. `http://www.doc.ic.ac.uk/~jwa/stage`.

[6] J. Ayres. Implementing Stage: the Actor based language. `http://www3.imperial.ac.uk/computing/teaching/distinguished-projects`, June 2007.

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

[8] T. Desell, K. E. Maghraoui, and C. Varela. Load balancing of autonomous actors over dynamic networks. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90268.1, Washington, DC, USA, 2004. IEEE Computer Society.

[9] M. Fowler. Fluent interfaces. `http://www.martinfowler.com/bliki/FluentInterface.html`, 2005.

[10] P. Haller and M. Odersky. Actors that Unify Threads and Events. In *International Conference on Coordination Models and Languages*, Lecture Notes in Computer Science (LNCS), 2007.

[11] 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.

[12] J. Lee. Python Actor Runtime LibrarY. `http://osl.cs.uiuc.edu/parley/`, July 2007.

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

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

[15] J. Sillito. Stage: exploring erlang style concurrency in ruby. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 33–40, New York, NY, USA, 2008. ACM.

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

[17] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994.