**MEng Project Report**

# Stage#

A Truly Distributed & Scalable Language

Christopher Zetter
cmz05@doc.ic.ac.uk

Supervisor: Susan Eisenbach

June 2009

# Abstract

The Actor programming model makes writing highly concurrent applications a possibility. However, existing actor-based languages fail to make it easy for the programmer to exploit the full level of concurrency possible across many processors and many hosts in a distributed system.

To solve this problem we modify the existing actor-based language Stage to create Stage#, a language which is built around a distributed hash table that is used for actor location.

We demonstrate how powerful the actor location model in Stage# is by using it to create generic solutions to solve common problems in distributed systems such as the distribution of work. By comparing Stage# to other actor languages we show its expressiveness and ability to easily create concurrent, scalable applications.

# Acknowledgements

First of all I would like to thank Susan Eisenbach for supervising my project and giving me valuable advice and guidance throughout my work. I also thank Sophia Drossopoulou for some useful discussions we had about actors.

Secondly I thank John Ayres for creating the original Stage, a language which has been incredibly well thought out and, during my use of it, has proved very robust. John has given me many useful suggestions and critique regarding Stage.

Also I thank Namit Chadha for several discussions about our actor-based projects, and I thank John Field of IBM research for providing me with information about the Thorn language.

Finally I would like to thank my parents for their full support while I have been at university.

# Contents

**1 Introduction** 1
1.1 Motivation . . . 1
1.2 Stage . . . 2
1.3 Contributions . . . 2

**2 Concepts of Distributed Systems** 5
2.1 The Actor model . . . 5
2.1.1 An Actor . . . 5
2.1.2 Actors work well in large distributed systems . . . 6
2.2 Code Mobility . . . 6
2.2.1 Uses . . . 7
2.2.2 Actor Mobility . . . 7
2.3 Resource location in distributed systems . . . 8
2.3.1 Broadcasting . . . 8
2.3.2 Gnutella, a Peer to Peer system . . . 9
2.3.3 Distributed hash tables . . . 10
2.4 Balancing distributed systems . . . 12
2.4.1 Strategies for load balancing . . . 12
2.4.2 Heterogeneous Networks . . . 13
2.4.3 Balancing Actors . . . 13
2.5 The Diversity of Distributed Systems . . . 14
2.6 Conclusion . . . 15

**3 Languages for Building Distributed Applications** 17
3.1 Erlang . . . 17
3.1.1 Features . . . 17
3.1.2 An Example . . . 18
3.1.3 Process Location . . . 19
3.2 Stage . . . 19
3.2.1 Migration . . . 20
3.2.2 Lazy Synchronisation . . . 20
3.2.3 Actor Classification . . . 21
3.2.4 Naming and Look-up . . . 21
3.2.5 Load Balancing . . . 21
3.2.6 Syntax and Comparison . . . 21
3.3 SALSA . . . 22
3.4 JXTA . . . 23

3.5 Conclusion . . . . . 23

4 Stage# 25
4.1 Managers and Theatres . . . . . 25
4.1.1 Identifying Actors . . . . . 26
4.2 Overlay Network . . . . . 26
4.2.1 Redundancy in the Network . . . . . 28
4.3 The Stores . . . . . 29
4.3.1 Names . . . . . 29
4.3.2 Types . . . . . 30
4.3.3 Migrations . . . . . 32
4.3.4 Source . . . . . 33
4.4 Summary . . . . . 34

5 Higher Level Concepts 35
5.1 Actor Ontology . . . . . 35
5.1.1 Singleton Actors . . . . . 35
5.1.2 Creatable Actors . . . . . 36
5.2 Dynamic Code Replacement . . . . . 37
5.3 Scaling in all directions . . . . . 38
5.3.1 Multicore . . . . . 39
5.3.2 Mobile Devices . . . . . 41
5.4 Distribution of Work . . . . . 42
5.4.1 Actor Pools . . . . . 43
5.4.2 Pool Invocation . . . . . 44
5.5 Visualising and Debugging Stage# . . . . . 45
5.6 Summary . . . . . 46

6 Evaluation 49
6.1 Scalability of Process Location Mechanisms . . . . . 49
6.1.1 Cost of storing information . . . . . 50
6.1.2 Lookup hops . . . . . 51
6.1.3 Cost of adding nodes . . . . . 52
6.1.4 Redundancy . . . . . 52
6.1.5 Planning . . . . . 52
6.1.6 Conclusion . . . . . 53
6.2 The language . . . . . 54
6.2.1 Comparison to JXTA . . . . . 54
6.3 Distribution Mechanism . . . . . 57
6.3.1 Adapting the Code . . . . . 57
6.3.2 Results . . . . . 57
6.3.3 Conclusion . . . . . 59
6.4 Yapper, A Distributed Micro-blogging Service . . . . . 59
6.4.1 Implementation . . . . . 60
6.4.2 Use of the Stage#'s Features . . . . . 60
6.4.3 A Scalable Architecture . . . . . 61
6.4.4 Conclusion . . . . . 61
6.5 Summary . . . . . 62

7 Conclusions and Further Work 63

7.1 Using a Distributed Hash Table for Process Location . . . 63
7.2 The Stage# Language and Applications . . . 64
7.3 Further Work . . . 64
7.4 Closing Remarks . . . 65

**A Storing Information** **67**
A.1 Conditions . . . 67
A.2 Test Programs . . . 67
A.2.1 Erlang Server . . . 67
A.2.2 Erlang Worker . . . 68
A.2.3 Stage# Server . . . 68
A.2.4 Stage# Worker . . . 69
A.3 Results . . . 69
A.3.1 Erlang . . . 69
A.3.2 Stage# . . . 69

**B Actor Pools Test** **71**
B.1 Conditions . . . 71
B.2 Results . . . 72

**C JXTA Client and Server** **75**
C.1 The JXTA Server . . . 75
C.2 The JXTA Client . . . 77

**D Yapper** **79**
D.1 Yapper `Client` . . . 79
D.2 Yapper `Person` . . . 80

**Bibliography** **81**

# Chapter 1

# Introduction

## 1.1 Motivation

Actors simplify the complex task of concurrent and distributed programming. Due to their lack of shared memory and the implicit thread every actor has, the programmer does not have to worry about the error-prone concepts of threading and locking. There is certainly a need for such a model; while the "concurrency is hard" mantra is heard repeated among despairing object oriented programmers the number of cores in the next mass-market processor line doubles. There is also a need for concurrency across many machines in areas such as web services and it is here the Actor model has recently simplified the creation of distributed applications [39] [65].

While the Actor model may allow a program to run on many cores across many machines the model provides no concession for managing actors[1] distributed across many hosts other than the message passing abstraction. In a dynamic system where actors are continually created, destroyed and move between hosts a method of actor location needs to be used to find other actors at run time. Many actor based systems provide inadequate solutions for this problem; for example, Erlang uses a broadcasting based location mechanism which does not scale to even a modest amount of hosts [52].

We claim that existing solutions for actor location, and indeed process location in any distributed language, are limited and lack intelligence. The actor-based languages Stage, Erlang and SALSA are limited because they only provide for the association of an actor to a name and are unintelligent because they do not ever automate the locating of actors. For example, when any actor migrates between hosts we would like all communication to that actor to be automatically sent to its new location. Existing languages do not attempt to deal with this behaviour, forcing the programmer to implement it themselves and create a solution that is not portable or scalable. We shall see that a better actor location model cannot just help with actor migrations but provide solutions for other problems common in distributed systems including the distribution of work and the dynamic updating of code.

[1]Here the uncapitalised actor is used to talk about a process-like entity as defined by the Actor model.

## 1.2 Stage

Stage is an actor based language written in Python by John Ayres [14][16]. Stage overrides the default behaviour of objects in Python to create a language that shares the Python syntax but uses asynchronous message passing instead of method invocation. Stage provides many additional features, including load balancing, futures & actor migrations, all of which are discussed in section 3.2.

The following `Hello World` program demonstrates what Stage does:

A Python Object:

```


class Hello(object):
  def hi(self):
    print("Hello world!")

hello_object = Hello()
hello_object.hi()
print("Method Invoked")
```

A Stage Actor:

```
from Actors.keywords import *

class Hello(MobileActor):
  def hi(self):
    print("Hello world!")

hello_actor = Hello()
hello_actor.hi()
print("Message Sent")
```

While they look very similar their semantics fundamentally differ. Importing `Actors.keywords` and inheriting from `MobileActor` rather than the Python base class `object` means that the `Hello` class on the right is no longer an object but an actor! It has its own thread and message queue. In Python performing `hello_object.hi()` will have the expected outcome (the thread executes the `hi` method). In Stage `hello_actor.hi()` causes a message to be asynchronously sent to the `Hello` actor. This asynchronicity means “`Message Sent`” may be printed before “`Hello World!`”.

We shall see that, just like other actor based languages, Stage has deficiencies in actor location and the distribution of work (Chapter 3). We reason that because of its relative simplicity and high-level implementation it provides a good starting point for a language designed to fix these problems.

## 1.3 Contributions

We extend the language Stage to create Stage# (pronounced ‘stage-hash’). The key addition to Stage has been the implementation of a distributed hash table to provide a cheap, distributed and scalable method for storing actor locations (and other information about the running system) with a configurable level redundancy (Chapter 4).

The rest of Chapter 4 explains the five separate data *stores* we build on top of our distributed hash table. The separate stores keep track of hosts, named actors, type of actors, migrated actors and the source of actors. Some of these stores have insertion/lookup functions available to the programmer while others stores are maintained automatically by Stage#.

As well as giving the programmer direct access to some stores in the distributed hash table we provide a set of higher level abstractions for the programmer to use including methods for distributing work across hosts and dynamic code updating (Chapter 5). These are implemented using the distributed stores so by showing their usefulness we, in part, justify the existence of our stores.

We aim for Stage# to be a language that works on any type of computer so in section 5.3 we explore the modifications made to Stage so it can be used effectively both on multicore machines and on low powered mobile devices.

Applications written in Stage# may be distributed across many machines at runtime. This makes debugging Stage# applications using existing machine-centric debugging tools difficult. In order to aid in the design of Stage# itself and debug programs written for Stage#, we create a tool that visualises the current state of an application running in Stage#. This tool, called the Visualiser, is described in section 5.5.

For the purposes of evaluation (Chapter 6) we create two Stage# programs; one that performs trapezoid approximation, and a distributed micro-blogging tool called Yapper. Both of these programs make full use of Stage# features– the distributed stores and the higher level abstractions.

# Chapter 2

# Concepts of Distributed Systems

First we cover a range of topics on the features and ideas common among distributed programming languages and the distributed systems they create. We describe many of these features, such as code mobility and load balancing, specifically in terms of the asynchronous message passing model of Actors even though they apply to many programming models. This is because it is the model most prevalent among distributed languages and is what Stage is based upon. The areas that we explore in the greatest depth, such as the problems of resource location and load balancing, are precisely the ones that (as we shall see later in section 3.2) the original version of Stage stands to be improved in.

## 2.1 The Actor model

The Actor programming model enables concurrency. Originally described by Hewitt, Bishop, & Steiger [35], every actor implicitly has its own thread of control and communicates asynchronously using message passing. This allows us to easily write actors that can continue with their own computation rather than to wait for remote computation to finish.

### 2.1.1 An Actor

Every actor is made up of a message queue that contains communication from other actors and its own thread of control. To describe the behaviour of an actor Ahga defines an actor as an agent that takes messages from its queue and then can [4]:

1. Send outgoing communications to other actors

   This outgoing communications entails a `send` primitive must exist within the language that requires a message and destination.

2. Change of behaviour that may cause the next communication received to be handled differently.

   This could be achieved by the ability to replace the current actor by another, or by keeping and modifying an internal state that changes how future messages are handled.

3. Create a number of new actors.

   This could be implemented using `new`.

### 2.1.2 Actors work well in large distributed systems

> [Humans] don't have shared memory, so why should programming languages? [10]
>
> *Joe Armstrong, Creator of Erlang*

Actors abstract away the individual host. Since actors cannot share memory, instead copying everything into messages, actors are not dependent on having access to the same memory as other actors. Assuming actors have a globally unique name, there is no distinction for the programmer between sending a message to an actor on the same host and one that is not. However this does add the complication of needing a global way to look up actors. In chapter 3 we shall see how the actor based languages Erlang, Stage and SALSA tackle this.

All this means that in general actors can be placed on any host. We will see when evaluating Stage that there are reasons for wanting an actor placed on a particular host. This exception is needed when actors are interfaces to non-actor components (such as drivers) that must be tied to a given machine.

## 2.2 Code Mobility

In a distributed system it can be useful to move executing code, such as objects or actors, between different hosts in the network. There are four common mobile code paradigms:

- *Client-Server*. The client requests a server to perform an operation. No code is transferred between hosts.
- *Remote Evaluation*. Code is pushed to a remote host to be evaluated.
- *Code on Demand*. Code is pulled from a remote host so execution can be performed locally.
- *Mobile Agent*. The executing thread moves from a local to a remote host.

Each one is shown in **Table 2.1** where A and B both are executing threads, *know-how* is the code that A needs to execute to perform its task and lastly a *resource* is something the know-how needs to complete. Resources can be seen as an immobile device the know-how uses such as processor time or a hard disk.

There are two classifications for mobile agents; *weak mobility* and *strong mobility* [21]. Strong mobility means any time during execution the actor may migrate, moving the its whole state, and resume from the point of migration. Since the actor could be midway through execution the stack must be moved too. This is a complex and expensive operation when compared to the alternative–weak mobility. After a migration in weak mobility execution always resumes at the same place, a 'start' method that must be provided by the programmer. This mitigates any need to copy the execution stack.

| Paradigm | Before | | After | |
|---|---|---|---|---|
| | Host X | Host Y | Host X | Host Y |
| Client-Server | A | know-how resource B | A | know-how resource B |
| Remote Evaluation | know-how A | resource B | A | know-how resource B |
| Code on Demand | resource A | know-how B | know-how resource A | B |
| Mobile Agent | know-how A | resource | | know-how resource A |

**Table 2.1:** Classification of the different mobile code paradigms [30].

### 2.2.1 Uses

A distributed application composed of mobile agents simplifies performing several useful actions at runtime. These actions include [30]:

- *Load Balancing*. By moving a computationally-bound actor from a machine with a high load to a comparable machine with a low load, computation will finish faster.
- *Performance optimisation*. Load balancing is one way to improve the performance of a distributed application. Alternatively, actors could be moved to improve other characteristics of the distributed application such as minimising the latency of communications.
- *Fault tolerance*. If a host is known to be going down all the actors on it will be able to transfer. If a host fails unexpectedly with no time for a process to 'jump ship' migration cannot be used. Redundancy needs to be put in place to recover from this.
- *Protocol encapsulation*. If data is formatted for a particular protocol it can be sent as a mobile agent with the code needed to access the data from the protocol. For a protocol that is tightly coupled to the code this may provide a nice solution.
- *Maintenance*. The ability to send code from one host to another means we can change the code executing on a host at runtime. This can be used for patching systems without stopping them from running or automate introducing existing code to hosts joining the system.

### 2.2.2 Actor Mobility

Actors can be sent from one host to another by virtue of message passing[1] This is a form of *remote evaluation* [21]. Some actors languages also follow the *mobile agent* paradigm by providing a `migrate` primitive. By looking at **Table 2.1** we can see how mobile agents are different from remote evaluation. When following the mobile agent paradigm the actor with its state is moved to another host, leaving nothing on the original host. However in remote evaluation the actor code is copied and the state is not. Agha explains that because communication lag is such a dominant factor of

[1] In a strict Actor model *everything is an actor* including messages just as *everything is an object* in the object oriented programming model. It is useful to define primitive types which are exceptions to this rule just as is done in many object oriented languages.

execution time in a distributed system of actors that process migration must "*...acquire a pivotal role in the efficient implementation of actor languages*" [4].

## 2.3 Resource location in distributed systems

When writing a distributed system it is not always known how it will be deployed– for example, the location of the host where a certain process will run. Furthermore, even if the deployment is known it is likely to change; more hosts may need to be added to provide more hardware resources for the system or hosts may fail and could need replacing. The run time entities on a host may change; for example, new actors could be created or migrated to another host. We use *resources* to describe anything we want to keep track of in a distributed system– this could be actors, data or hosts. We use the term *dynamic* to describe a changing system in terms of the hosts and the actors running. Dynamic systems need a way of locating these changing hosts and actors if they're resources are to be used by other actors.

The server model is one way of locating resources. The location information of resources are registered with the server and then retrieved when required. The problem with this is that it precludes servers from also being dynamic entities in the system unless a means to locate servers is provided. Servers also require planning and administration from a centralised authority. Because of this we instead focus on more decentralised mechanisms for resource location.

### 2.3.1 Broadcasting

In the broadcasting model information of each resource is sent to all hosts in the system. Each host keeps a local store of all of their collected information and looks up resources locally. This is a very simple model, multicasting improves on this by using a special type of packet to reduce outbound traffic sent from a host.

#### Multicasting

Multicasting is an addition to the Internet Protocol Suite that allows broadcast routing. Normally a packet is sent to a single destination (unicasting). When packets marked as multicasted reach a router they are forwarded on every link rather than just one [24]. Two examples of systems that successfully use multicast is the device & service discovery protocol Bonjour [7] and the Java distributed programming framework Jini [59]. The use of multicast in these systems is thus;

1. whenever a new process wants to join the network it will send a multicast message asking for process registries.
2. Any process registry that hears the message will respond with information about its location.
3. Now the process can send a direct message to the registry asking to be stored.

If a process is searching for another process it can ask for registries in a similar way. This method has the benefit of making it very easy to add another registry to the system; it will hear the same multicast messages as other registries and can respond to them in the same way. This is desirable since having only one centralised registry creates a single point of failure. Multicasting is blocked by many routers that make up the Internet so is not supported between any two hosts in most cases.

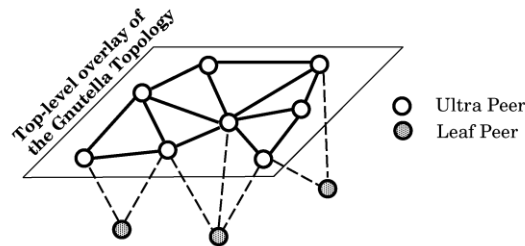



**Figure 2.1:** The overlay produces interconnected networks [55].

Instead multicasting can be used on local networks. It is possible to use tunnelling to connect two networks to guarantee that multicast communication can be sent between them [45] or fall back to ordinary broadcasting.

### 2.3.2 Gnutella, a Peer to Peer system

Gnutella is widely used protocol for large peer to peer file sharing networks. Gnutella has an open specification allowing us to examine how it has evolved to fix scaling problems in the original client.

#### Gnutella's Scaling Problems

There are a few problems with the Gnutella architecture. It has and overlay network that is constructed in an ad-hoc manner, nodes joining other nodes with no regard for the topology of the physical network. This results in a topology mismatch between the physical network and the overlay [42] ultimately causing a sub-optimal structure to be created. Object look-up is not guaranteed; one node could search for an object and will never find it even if it exists in the network. This would be a terrible characteristic to have in Stage. An example which illustrates this is thus; in Stage we can have singleton actors which are unique across a whole network. If a singleton actor is needed but cannot be found there is no sensible way to proceed since you cannot create a singleton if you do not know if one already exits. Ritter discusses the problems Gnutella has scaling [47]. Due to the use of 'flood searching' queries will create a lot of network traffic. If the network is optimally constructed, a search of bytes in a network of millions of nodes will create traffic in the order of megabytes. In a more likely sub-optimal structure it is possible for gigabytes of traffic to be created as the tiny search term is copied many times along the physical network for it to reach nodes in the overlay network.

#### Gnutella *can* scale

There have been many forks of the original Gnutella project, most notably Gnutella2, attempting to fix these issues. The Dynamic Query Protocol [26] is implemented in Gnutella2 as a replacement for flood searching. First an estimate of the popularity of the file is calculated by doing a probe query that has a limited search depth and then the depth of the search is altered so the query terminates when a maximum number of results have been returned. If the query is rare it still will be propagated many times resulting in a lot traffic just like a flood search. Newer iterations of the Gnutella protocol have nodes which are dedicated to routing called ultrapeers. These ultrapeers

result in a more interconnected network which decreases the number of hops it takes to reach other nodes, this is illustrated in **Figure 2.1**. A very interconnected network helps against node failures. Stutzbach et al. determined that Gnutella networks are incredibly resilient, predicting that after randomly removing 85% of nodes 90% of nodes will still be connected [55].

#### Conclusion

The set of Gnutella protocols have evolved to enable resilient peer to peer networks that are good at finding resources which there are many copies of in the network. However due to the lack of any guarantee of being able finding any one file Gnutella is not suited to finding resources that are located on a small number of nodes.

Due to the unstructured nature of the network the topology of a network overlay will vary and we cannot reason about characteristics of an arbitrary sized Gnutella network. Instead crawlers can be made to discover the topology of a specific Gnutella network and then can be used to estimate actual properties [55].

### 2.3.3 Distributed hash tables

Distributed hash tables (DHTs) have been a popular area of research recently. They provide a completely decentralised network that can locate resources efficiently. CAN, Chord, Pastry and Tapestry [46][54][51][66] are all DHTs that share the same underlying ideas.

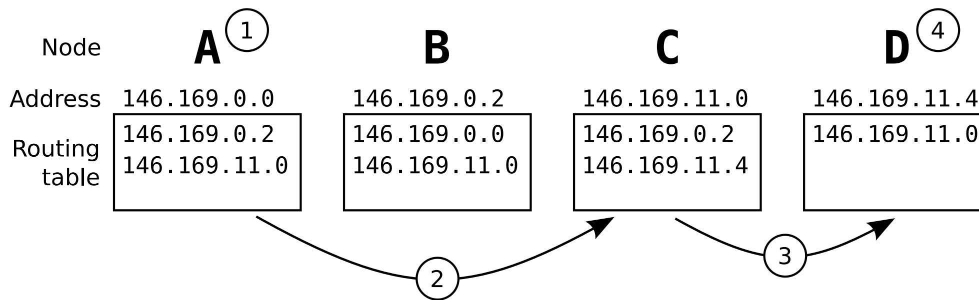
#### Properties

In a DHT a hash of the object wanted to be stored is taken. The object is then stored at the node responsible for that hash. Responsibility is a mapping of hashes the node IDs which are assigned to every node; so by knowing the hash of the object you know the node ID and can route to this node by use of the routing table at every node. Unless the network has a number of nodes close to the number of possible node IDs, it is likely the node ID mapped from the hash does not exist in the network. To get around this problem all routing in the system is done by forwarding any messages to the node ID that exists in the routing table that is 'closest' to the destination of the message. If the message reaches a 'dead end', meaning that it is not possible to forward the message to another node, then it is the node it is currently at that is responsible.

Thus there are two primitive operations that can be performed on a DHT;

1. `put(key, object)` where `key` could equal `hash(object)`. This stores the data in the network.
2. `get(key)` this retrieves the object, with the specified `key`, from the network.

For `get` to be useful the hash of the object we are trying to find must already be known. The unique name of an object could be hashed rather than its contents to give the same properties.




1. **A** performs `hash(x)` to produce key `k`. If `k = 146169011006` this will map To address `146.169.11.6`
2. **A** forwards `(k, x)` to **C** since `146.169.11.0` is the closest address to `146.169.11.6` in its routing table
3. **C** forwards `(k, x)` to **D** since `146.169.11.4` is the closest address to `146.169.11.6` in its routing table
4. **D** has no entry in its routing table that is closer to `k` than its own address so it stores `x`.

**Figure 2.2:** A node stores the object `x`

#### Example

**Figure** 2.2 shows how an object is stored in a simplified DHT network with four nodes and a simplified hash that maps objects to IP addresses. In practice DHTs usually pick node IDs which are independent of IP addresses so they are evenly distributed throughout the range of possible node IDs– this is important to evenly distribute objects across nodes. Object retrieval happens in a similar way; If **A** was looking for the same object `x`, with the hash `k`, it would forward the request with `k` to **C**. The request will eventually reach **D** which will have `x` if it exists.

#### Efficiency and Redundancy

Depending on the number of nodes in each routing table and their node ID distribution and upper bound for a look up action can be calculated. For example Pastry splits up a 128-bit node ID in to a set of numbers, each $b$ bits long, and nodes have levels in their routing tables for each matching set of bits. Each level stores the same amount of nodes but higher levels are taken from a smaller set of nodes with more matching sets. As a result any pastry node will know a large proportion of nodes ID that are very numerically close to its own node ID and differ only by one set, and only a small proportion of total nodes that differ by all sets. Using this system any action in Pastry has $O(\log n)$ routing steps. While differing, among other things, in the structure of the node ID and the organisation & size the routing table most DHT systems have this same upper bound for routing [51].

For the system to be able to cope with a node failing the stored data must have redundancy across nodes. The can be done by adding constant ‘salt’ values to the key to produce a set of keys [66]. The object is then stored at each of these locations. Now for `get(key)` actions there will be more than one `key` value to choose from. If nodes have a sense of what nodes in their routing table have the quickest response time they can choose a `key` that will likely result in the quickest possible `put` action.

**Uses**

Distributed hash tables have found their way into many distributed systems, one example is the Coral Content Distribution Network[2] that uses DHT for look up. Their network of hundreds of machines globally has made a great test bed for DHTs. One problem that Coral found is the assumption of transitivity in the network i.e. all nodes can communicate with all others. Due to the complicated structure of some networks this may not be the case and assuming it is may cause routing requests to get stuck or, in the case of Chord which has a circular overlay, loop around the network. The solution is for nodes to only add (and remove) entries from their routing table that they have tried reaching themselves so they are not solely trusting information from other nodes [28]. A similar problem exists if a `get` request is recursively routed around the network from one node to another and then takes a direct route from the node storing the object to the originator of the request, a route that may not be possible. Acknowledgements and timeouts are needed so alternative routing paths are known when to be used.

Another interesting and relevant use of DHTs is Scalaris [52]. Scalaris is an Erlang based implementation of a web server for Wikipedia[3] that stores all of the vast Wikipedia in the pooled main memory of many machines. A DHT is used to find in which machines memory the needed resources are stored. Using a DHT also makes redundancy easy to implement by use of a technique called symmetric replication (similar to the salt technique it translates a single key to a set of keys). It is the use of this peer to peer like architecture that makes it easy to add and remove new nodes to the system making it scalable and robust.

## 2.4 Balancing distributed systems

Balancing is making best use of resources available to complete a task in the minimal execution time possible. Balancing is achieved by moving processes from one node to another. A process performing this task of moving processes, or *load balancer*, needs to decide what process to move and what node to move it to.

### 2.4.1 Strategies for load balancing

Here we look at some strategies for load balancing as discussed in [60] in the context of dynamic distributed systems.

- *Sender Initiated Diffusion*. Nodes maintain communication with their neighbours so they know their loads. A node that is overloaded will move processes to underloaded neighbours. If a whole area of the system is overloaded, processes in this section will eventually *diffuse* to underloaded parts. This has previously been implemented with success in agent based systems [53]. Many grid-based systems that have analogous schemes have a centralised store of resources available in the system (i.e. underloaded machines) and then assign work to these resources. There some grid systems that store advertisements for resources in a distributed hash table to avoid the centralised store and have seen a performance improvement from doing so [56] [20].

[2] The Coral network provides a distributed cache of websites - `http://www.coralcdn.org`

[3] Wikipedia is of course everyone's favourite online encyclopaedia - `http://wikipedia.org`. Unlike other popular websites Wikipedia has an open architecture allowing tinkering.

- *Receiver Initiated Diffusion*. Similar to Sender Initiated Diffusion but underloaded nodes pull processes from their overloaded neighbours. This has the nice property that only underloaded nodes have to handle the load balancing overhead.
- *Hierarchical Balancing Method*. Nodes are organised into a tree denoting the hierarchy of the system. Every node is responsible for balancing the nodes below it in the hierarchy. This results in a centralised system.
- *Gradient Model.* Every node has a *proximity number* that says how far away that node is from an underloaded node. This is 0 when the node is underloaded, or an increment of the lowest proximity number from neighbouring nodes. An overloaded node will transfer processes to a neighbouring node that has the lowest proximity number, in other words its neighbour that is closest to an underloaded node. In this simple form of the Gradient Model it can take time for proximity numbers to propagate through the network causing too many processes to be transferred to underloaded nodes. To stop this sudden overloading of nodes an extension is given in [43] that uses a state machine that forces the incremental loading of nodes. Nodes that are underloaded and are about to receive a process change state. While in this state they will not accept any other transfers. After receiving the process and updating the load of the machine the state is changed back to accept new processes.
- *Dimension Exchange Method.* Nodes are paired with a neighbour and then balance between themselves. This is repeated for every neighbour the node has. This balancing act is synchronous so needs some coordination to work through the whole system. It is best suited to a hypercube topology because, as hypercubes are uniform, every node has the same number of neighbours. It is worth noting that some DHTs have a hypercube based overlay to provide efficiency and resilience in routing [32].

### 2.4.2 Heterogeneous Networks

The algorithms defined in the previous sections all assume that every node in the network is the same. This is unlikely to be the case. The performance of individual nodes will differ and so will the time to communicate between any two. Adding extra nodes to a network with load balancing that does not take into account these differences can have a measurably negative impact on performance [19]. Knowledge of a node's architecture or results from a benchmark must be used in order to weight its performance. As in [49] it is possible to adapt the diffusion load balancing model to take into account both computing speed for nodes and communication time between them.

### 2.4.3 Balancing Actors

There is another way to decide what actors are slowing down the system and would be good candidates to move to another machine– the length of an actor's message queue. Desell, Maghraoui & Varela look at this possibility, creating the notion of *actor satisfaction* [25]. An actor is said to be not satisfied if it cannot process messages as fast as it is receiving them. This is a vitally important idea from queuing theory; if the arrival rate of a queue is greater than the its throughput the queue length will keep increasing and lead to an unstable system. They propose that satisfied (underloaded) theatres should seek out unsatisfied (overloaded) actors.

This idea is also extended by monitoring communication between actors to try to keep tightly coupled actors together and estimating the performance of the nodes the actor is currently at compared to

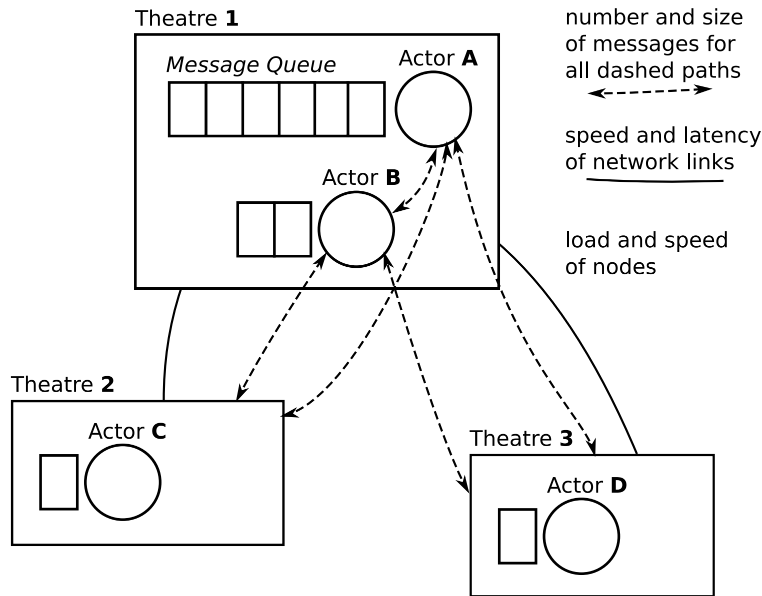



**Figure 2.3:** Balancing considerations.

the one it plans to migrate to. The example shown in **Figure 2.3** highlights some important metrics to be considered when attempting to move actors from the overloaded theatre **1**. If the volume of messages between actors **A** & **B** are large it might be beneficial to move them together. The load balancer must make a judgement on if the machine they are to migrate to is 'faster' than the current host to avoid migrations that negatively affect performance. One machine could be considered faster than another by comparing ratings from a benchmark.

Not explained in the system described in [25] is how to tell if a long queue length is not indicative of high processor load but due to the actor being blocked by another. In this case, migrating the actor that is blocked will not help and instead more importance should be placed on making sure the blocking actor is executing as fast as possible.

## 2.5 The Diversity of Distributed Systems

Distributed systems perform tasks using the resources of many machines. So far we have seen how the Actor model lends itself to distributed system programming because it has message based communication and enables concurrency. In the next few sections we will be looking at many distinct distributed systems that exist for totally different purposes. Here are some fundamental properties that can influence the implementation of the system:

- *Network Topology and Protocol*. How processes communicate? Stage and Erlang use an IP network for communication. In [67] the TCP/IP libraries of Erlang are replaced to speed up communication in a parallel machine with 128 processors.
- *Organisation of the System*. Are all nodes equal? Or is there a hierarchy of nodes? If some nodes are more 'trusted' than others they may be given different tasks. For example BOINC[4]

[4]BOINC (Berkeley Open Infrastructure for Network Computing) is a tool developed to allow volunteers to contribute

allows the distribution of the same work using trusted centralised servers to more than one client. This redundancy prevents any one person from tampering with results. There is no redundancy used for the trusted centralised servers [6].

- *Properties of Nodes*. Are nodes highly transient leaving and joining the system often as in Peer to Peer network or is the system made up of fixed reliable machines? If nodes are transient they must be able to join and leave the system at run time.
- *Purpose of System.* What are the nodes sharing? The system could have a computational goal so nodes share processor cycles or could be sharing files (such as the file sharing network Gnutella in section 2.3.2). The BOINC system is built to allow both.

The extreme ends of each of these properties can often be seen when comparing Peer to Peer systems to Grid Computing. Peer to Peer systems have been defined as running at the 'edges of the Internet'. They are made up of many nodes under individual control, the transient nature of these nodes leads to the total decentralisation of the system. In contrast the groups of nodes that make up Grid Systems are controlled by a central authority. Nodes are trusted and there is not as great a need to decentralised resources. In [27] Foster and Iamnitchi discuss this difference and how it has lead to the separation of these two types of distributed systems. They predicted that these two areas will ultimately approach a distributed system *"nirvana from different directions"*. Grid systems will need to deal with node failure and decentralisation as well as peer to peer systems do before they become truly scalable. Peer to Peer systems need to have the standardisation of protocols and service that are common in grid systems and the ability to perform as well on the structured networks seen in grid systems before they can be considered to be deployed in place of them.

## 2.6 Conclusion

Grid based and planned systems are very ridged. They run on dedicated machines on local networks and will have centralised resources for managing computation. Peer to peer systems can provide a much more flexible solution to distributed computation. They allow machines to leave and join the network. For example, DHTs are dynamically constructed so allow hosts to join and leave them at any time but still provide a fast and efficient means for resource location.

We have also seen how some systems, such as DHTs, are inherently *scalable* and some, such as Gnutella, are not. By scalable we mean that it is easy for the system to handle more work (which could be storing information or performing computation) by adding more hosts to the system. Any extension to Stage that we create should both be scalable itself (i.e. the feature should not negatively affect performance for a large number of hosts) and facilitate the construction of scalable applications.

computing resources to one of many projects including SETI@home which distributes the analysis of radio transmissions in search of alien life.

# Chapter 3

# Languages for Building Distributed Applications

Now we examine three actor based languages, focusing on their features. We also look at JXTA which, while being a protocol rather than a language, is still relevant to Stage and Stage#.

## 3.1 Erlang

Erlang is a functional language that follows the Actor model for concurrency. Originally Erlang was developed by Ericsson specifically for telephony applications [9]. Since 1998 it has been maintained as an open source project. As well as still being popular in telecommunications Erlang has recently gained favour with developers outside this original domain, for example the chat system in the popular Facebook uses Erlang and so does Amazon for some applications. In these systems there is a shared need to handle many concurrent users (in the millions) in a quick and reliable way [39] [65].

### 3.1.1 Features

The features Erlang boasts are discussed here [12]. Some are a direct consequence of the Actor model, others are implementation choices.

- *Distribution.* Every node in the network runs an Erlang virtual machine. This virtual machine can create parallel processes to run on other nodes. Processes residing on different nodes communicate in exactly the same way as processes on the same node do (described in section 2.1.2).
- *Concurrency.* Erlang has its own lightweight processes that are operating system independent. As with any actor system processes have no shared memory and communicate by asynchronous message passing. Erlang supports applications with very large numbers of concurrent processes. Process creation in Erlang is over an order of magnitude faster than in C# and Java [8].
- *Robustness.* Erlang has error detection primitives built in that can be used to create fault-tolerant systems. Processes can be set up to monitor the status of other processes allow them,

for example, to perform an action if the process exits abnormally. Processes can be configured to migrate away from their current node if a failure occurs and then automatically migrate back to recovered nodes.

- *Soft real-time.* Erlang uses incremental garbage collection to avoid any long delays that would be undesirable in a real time system.
- *Hot swapping of code.* In Erlang code can be changed in a running system. This allows upgrades and bug fixes on a running system without any downtime.

Erlang also has a useful set of libraries and provides interfaces to other languages such as C.

### 3.1.2 An Example

**Listing 3.1:** Erlang Area server [9].

```
1  -module(area).
2  -export([loop/1]).
3  loop(Tot) ->
4    receive
5      {Pid, {square, X}} ->
6        Pid ! X*X,
7        loop(Tot + X*X);
8      {Pid, {rectangle,[X,Y]}} ->
9        Pid ! X*Y,
10       loop(Tot + X*Y);
11   end.
```

**Listing 3.2:** Erlang Area client

```
1  Pid = spawn(fun() -> area:loop(0) end),
2  Pid ! {self(), {square, 10}},
3  receive
4    Area -> Area
5  end.
```

In **Listing** 3.1 and 3.2 we can see some communication between two simple Erlang processes. The client creates the server and stores the resulting Pid in a similar style to a Unix `fork()` (line 1). However unlike Unix process identifiers, Erlang process identifiers are globally unique in a network. They are constructed by a unique node number, a process identifier unique to that node, and a 'creation' counter that will increment if a node is restarted to lessen the chance of temporal clashes of identifiers [2]. In line 2 of the client the '!' notation is used to send a message to the server process. This message contains the Pid of the client so it can be called back from the server with the total (line 6 and 9).

### 3.1.3 Process Location

Erlang provides a `register(Name, Pid)` function to bind process identifiers to names for a single node. After this binding a process can be referenced by name (which is known at compile time) rather than by Pid (only known at runtime).

For a network of nodes setting up every node can be difficult. Every Erlang node must be first named in the format `name@host`. This can be done when the Erlang runtime is started, for example the node named `deepthought@edge01` can be created by:

```
edge01$ erl -sname deepthought
```

Next every other node that is to be in the same network must have some initial communication with a node in the network so it knows that it exists. This can be achieved by a ping:

```
(deepthought@edge01)1> net_adm:ping('marvin@edge02').
```

Since registered processes have to be unique in the network and nodes can enter the network with many processes already running there has to be a way to deal with name clashes potentially created when a new node joins. In Erlang one of the processes is unluckily chosen and then terminated by the virtual machine.

In order to know where to send messages every Erlang node maintains a list of all other nodes in the same network. The function `register_name(Name, Pid)` can be used to bind names to process identifiers in a network of nodes [2]. This is implemented by the existence of a copy of the global name table at every node. Any time a new process is created, destroyed or moves hosts it must update every table of every node in the network. In a large system with many transient nodes or migrating processes a great deal of network traffic will be created. While experimenting with large Erlang systems Schütt, Schintke, & Reinefeld observed this problem, complaining that for large deployments in Erlang that *"...the network traffic caused by the management tasks within the VM dominates the overall traffic"* [52].

## 3.2 Stage

Stage is an actor based language created by John Ayres [14][16]. It is built on top of the multi-paradigm language Python[1], sharing common syntax with it. One of the justifications for the creation Stage was due to the state of other actor based languages, criticised for having "obscure syntax, immaturity, poor scalability and poor performance" by Ayres.

At every node in the network the Stage execution environment or *theatre* runs. In this theatre the actor scripts[2] that define actors run. It is theatres that manage the actors and handle communication between them rather than the underlying operating system. These parts of Stage are all shown in **Figure 3.3**. The *meta hooks* shown in the diagram are the method though which Python allows its own calling conventions to be subverted, as we see in **Listing 3.4** this has the result of allowing a familiar '.' syntax of Python (as well as C++ & Java) to be reused for sending messages. Stage certainly is a true Actor language– messages are passed asynchronously from one actor to the queue of another. Other features of Stage are evaluated in the subsequent sections.

[1]Python has features common to both object based and functional languages
[2]Note an *actor script* is the Python code that defines the actor while and *actor* is that code executing in a theatre

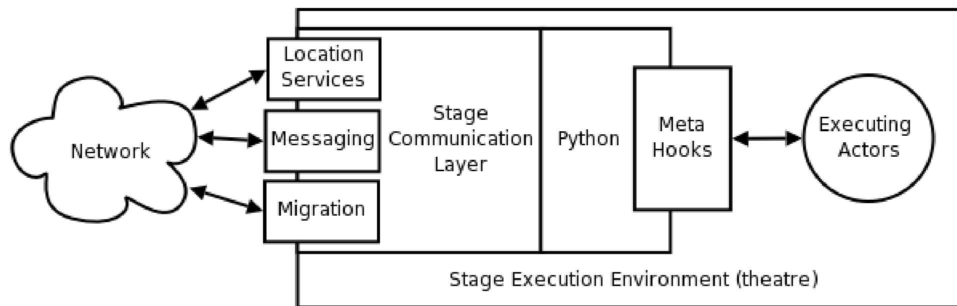



**Figure 3.3:** A stage execution environment (theatre)

### 3.2.1 Migration

There is a `migrate_to` primitive in Stage that allows weak migration. An `arrived` method is always executed after migration allowing the actor to re-perform any set up actions. By design an actor can only migrate itself. If you wished to design a system that had a load balancing actor, whose task was to signal other actors to migrate, you would have to make sure every actor in that system could migrate when signalled and have an implementation of the `arrived` method. This leads to a lot of repeated code, the type the clear syntax of stage was to avoid.

### 3.2.2 Lazy Synchronisation

Stage has implicit synchronisations. In **Listing 3.4** the `oven.grill(bacon)` statement and `breadbin.getslices(2)` can be performed in parallel since the assignments do not block, instead each method returns a *future*.

**Listing 3.4:** Lazy Synchronisation in Stage

```
1 bacon = fridge.getrashers(2)
2 bread = breadbin.getslices(2)
3 cookedbacon = oven.grill(bacon)
4 sandwich.create(cookedbacon, bread)
```

Futures were first discussed by Baker & Hewitt in [17] and are certainly not unique to Actors. Indeed since version 1.5 Java has had an interface that helps to write futures[3] Unfortunately the Java syntax is not easy to understand as it is not built into the language as it is with Stage.

When a future is accessed, as `sandwich.create()` does, it blocks until its true value is found. Because no blocking occurs until the last possible moment this achieves the most parallelism possible without the programmer even having to think about it. So futures can be seen as a place holder of where a result will be stored. During the development of Stage the choice to include futures was not easy. It is possible to create a system of actors with a circular dependency, each actor waiting for the result from another. This system will become *deadlocked*; execution will be stuck with no chance of any progress being made. Because synchronisations are implicit when futures are used, debugging a state of deadlock may be harder for a programmer. After programming in Stage with

[3] Java future API - http://java.sun.com/javase/6/docs/api/java/util/concurrent/Future.html

and without futures, Ayres concluded that the *"extra conciseness and clarity afforded outweighs the dangers of implicit synchronisation"* and found that deadlocks caused by futures could be resolved quickly.

### 3.2.3 Actor Classification

Stage has a few actors with useful properties built in. Using inheritance new actors can also benefit from these properties. Using inheritance in an actor based system is not a new idea [36] and is certainly as justifiable and useful as it is within object oriented languages [38]. In Stage a `MobileActor` can be considered the most generic type of actor. A `LocalActor` is a specialisation of a `MobileActor` that cannot move hosts. This is useful if it using a host specific resource such as the file system. There are also is a `LocalSingletonActor` and a `NetworkSingletonActor`. These only allow one named actor on a host and on the network respectively.

### 3.2.4 Naming and Look-up

To communicate with another actor Stage starts looking for the host where it thinks the actor should be. If the actor is not there Stage will keep asking any other hosts it is aware of if they know the location of the actor. If migration has occurred 'hints' can be left at previous hosts saying the new location of the actor. This is not a viable solution if the previous host is no longer available which is a likely situation since one of the motivations for code mobility is to survive machines shutting downs. Nor is it a solution that can provide a formal guarantee of actor location. This pushes the burden of keeping track of actors onto the programmer. We examined other distributed systems in search for a better alternative in section 2.3.

### 3.2.5 Load Balancing

A `ProbingLoadBalancer` is included in Stage and it operates on the principle of Sender Initiated Diffusion. While it only is meant to serve as an example for load balancing and migration it has been proven to work in some situations. It probes the processor load of know nodes and attempts to migrate actors from an overloaded machine to an underloaded machine. It does not take into account any other metrics, such as actor communication. Two actors that have a large amount of communication will be better to keep on the same node. Section 2.4 is concerned with looking at proven systems of balancing.

### 3.2.6 Syntax and Comparison

**Listing 3.5** & **3.6** shows actor communication in Stage in an example analogous to the Erlang one shown earlier. In Stage actors can store their state across processing different messages in variables whereas in the functional Erlang recursion is used to remember the total. The syntax of Stage is essentially unchanged from Python but the semantics differ. The '`.`' notation of Python is overridden so it sends a message rather than invoking a method, for an example of this message passing in use see line 2 of the server.

**Listing 3.5:** Stage Area server

```
class Area(Actor):

  def square(x)
    area = x*x
    self.total += area
    return area

  def rectangle(x,y)
    area = x*y
    self.total += area
    return area
```

**Listing 3.6:** Stage Area client

```
areaServer = Area()
print areaServer.square(10)
```

## 3.3 SALSA

Varela & Agha created SALSA (Simple Actor Language System and Applications) specifically to run across large distributed environments such as the Internet [58]. It has a Java-based syntax and shares many features that we have already examined such as asynchronous message passing and the ability to migrate actors. What is distinct in SALSA is the naming and look up system for actors. SALSA has two ways of identifying actors:

1. *UANs (Universal Actor Names)*. These are globally unique names which give the name of the actor and the centralised server that always knows its location. A `bind` action is used to register an actor at a given UAN server. UANs are in the format

   `uan://wwc.server.com/anactor`

   where `wwwc.server.com` is the hostname of the UAN server.

2. *UALs (Universal Actor Locators)*. These are in the format

   `rmsp://wwc.host.com/anotheractor`

   where `wwc.host.com` is the hostname of where the actor is currently located. UALs will not work if an actor has migrated to a different host.

This system adds unnecessary complexity to the programmer who now has to worry about the differences between UANs & UALs and the separate methods SALSA has for performing operations on each of them. Perhaps even worse is the use of centralised servers for UALs creating a single and point of failure to the entire system.

## 3.4 JXTA

JXTA is a set of protocols developed by Sun to allow building peer to peer applications [31]. Sun also has a Java library that implements JXTA, so while JXTA is not a language itself it provides constructs to the programmer, such as messages, that can be used to build a distributed application. JXTA's relationship to Java is similar to Stage's relationship to Python– both bring features to the older language to make it easier to build distributed applications. The difference is that Stage is not just an add-on to Python but actually changes the behaviour of parts of Python.

JXTA describes an overlay network made up ordinary nodes and *rendezvous* nodes. Rendezvous nodes play a similar role as to ultrapeers in Gnutella; they are meant to be less transient nodes that form a stable overlay network. Originally this overlay network used broadcasting to share information but in the latest version of JXTA this overlay is based on a Distributed Hash Table (DHT) [57]. This DHT is used to store *advertisements* which can contain, among many other things, the locations of services.

Unfortunately JXTA requires a large amount of code to be repeated (also known as 'boilerplate code') to program even simple tasks; for example, the frequent task of message sending requires multiple lines just to construct the message. This is an operation that Stage already does automatically. The result is that even small applications require very verbose code. For an example of this verbosity in a small application please refer to the chat application in *JXTA, Brendon Wilson*, whose code occupies pages **405 to 441** [61]. So much of this application is concerned with managing items that are common to every distributed application (such as the structure of the network, the creation of advertisements and messages) that the actual 'chat' part is hidden. This does not help programmers understand code which leads to longer development times, higher maintenance cost and a greater chance of defects.

## 3.5 Conclusion

Every language we have examined provides a different model for finding actors; Erlang uses broadcasting, Stage a system of 'hints', SALSA a DNS based naming scheme and JXTA uses a Distributed Hash Table (DHT). Our knowledge of these models from Chapter 2 tells us that only DHTs can be considered scalable. This is why we choose to implement a DHT in Stage and unlike JXTA, which was not originally designed with a DHT, we wish to allow the programmer and the Stage virtual machine to take full advantage of the properties of our DHT while maintaining the clear and Python-inspired high-level syntax of Stage. We re-visit Erlang and JXTA in our evaluation (Chapter 6).

# Chapter 4

# Stage#

In this chapter we present our main addition to Stage; A method for actor location in a distributed system. We aim to:

- Provide a cheap, scalable and distributed store that can be used for actor registration and location. We build this store using a distributed hash table.
- Explore what information we should store to best serve the programmer in actor location.

We call this new system Stage#.

The majority of Stage#'s syntax remains unchanged from Stage; for example, message passing remains identical from the programmer's perspective. As a consequence of our new way to register and find actors in the system the behaviour of Stage has changed; notable examples include the change to message passing to guarantee finding actors that have migrated between hosts, in section 4.3.3, and the storing of actor type information at actor creation, section 4.3.2. Throughout this chapter we also introduce the new low level operations available to the programmer that allow actor registration and location.

## 4.1 Managers and Theatres

The two main components of Stage# are shown in **Figure 4.1**; the theatre, which contains executing actors, and the manager. The manager component has been completely re-written to provide a cheap means for actor registration and location across many nodes, achieved by using the managers to create a Distributed Hash Table (DHT). Theatres retain their original purpose of providing an execution environment for actors with some modifications. There has been additions to the list of services that theatres provide to actors including an interface to the new low level services of managers and new methods that are higher level encapsulations of manager services (such as distribution of work). Every theatre is given the address of a manager when starting and uses this manager as an interface to the DHT.

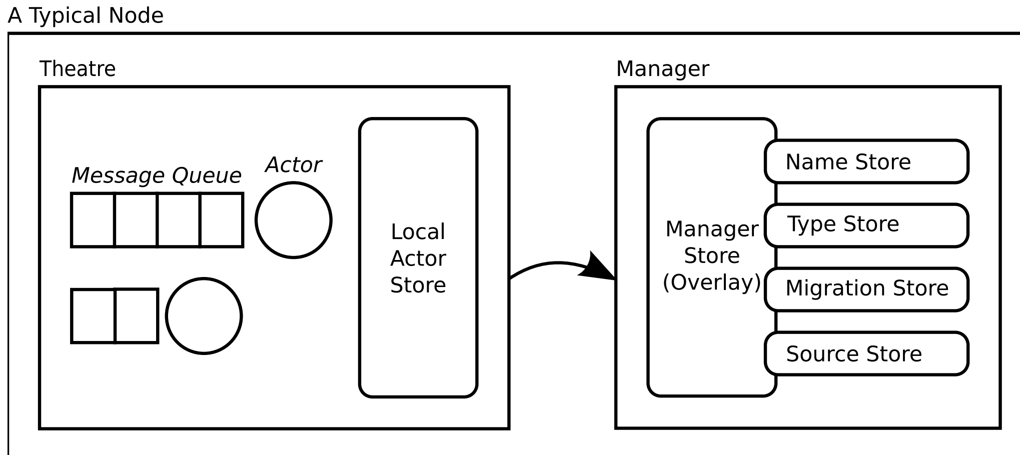



**Figure 4.1:** A node in Stage# has two components; a theatre and a manager.

### 4.1.1 Identifying Actors

We use an actor_id to uniquely identify an actor within a system. For example

146.169.51.70:7000_Worker_34

is an actor_id. It is composed of the location of the theatre in which the actor runs, the type of the actor and an identifier to distinguish it from other actors of the same type running on the same theatre. Knowing a given actor's actor_id is the necessary condition to sending that actor a message.

Every theatre needs to move incoming messages, addressed with an actor_id, to the memory address of the relevant actor's message queue. For this purpose a theatre has a local actor store that is defined by the mapping

$$\textit{actor_id} \longrightarrow \textit{address of actor's message queue.}$$

The address cannot be stored in the actor_id directly as it is under the control of the theatre virtual machine and has no guarantee of remaining constant over time.

Managers and actors all use the currency of actor_ids to identify actors. We will introduce some key/value stores which will often be concerned with translating a key, such as a name or a type, into an actor_id. This results in a crucial difference between all other DHTs and Stage#– other DHTs are built for finding data, while actors in Stage# want to find actor_ids to provide the means for message passing. This has influenced the design of our DHT; both on the construction of the overlay network and on what key/value pairs we store in it.

## 4.2 Overlay Network

We use a DHT which is primarily based on Chord [54]– it shares the same the overlay network and routing algorithm. Chord was chosen due to its relatively simple routing algorithm when compared to other DHTs and its support for replication [33].

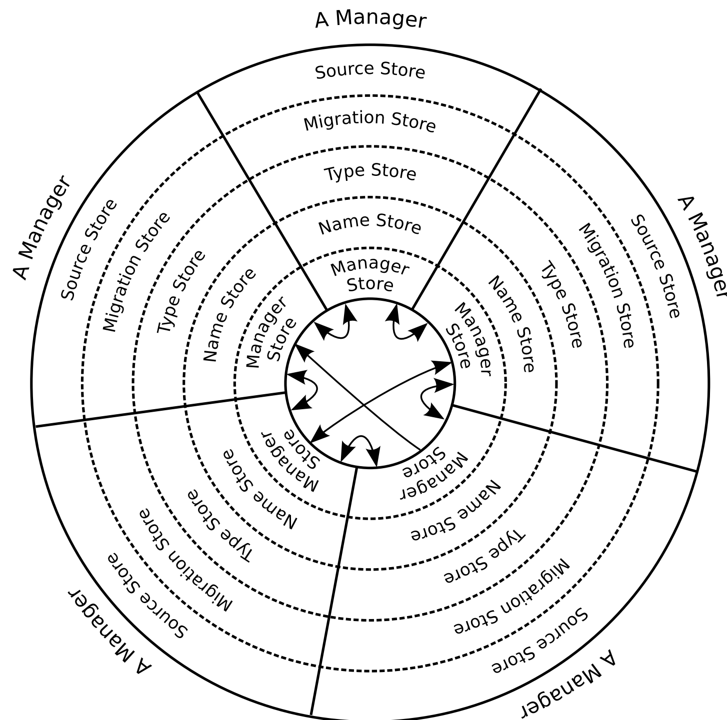



**Figure 4.2:** Managers combine to make logically contiguous storage spaces.

To form the DHT there must be some ordering on managers. On joining the system the manager generates a random number within the hash space of the DHT to use for a manager_id. This idea is taken from Pastry [51] and was chosen for its simplicity over more advanced hash space partitioning schemes such as the zone joining and splitting in CAN [46]. We found that for the majority of test cases random numbers were uniformly distributed enough to create a sensible overlay network. Those that were not sensible networks, i.e. those containing managers that were responsible for very different proportions of the hash space, could always be fixed by adding more managers (due to the law of large numbers).

Managers form a ring based overlay network for the DHT. The manager store within a manager holds the locations of other managers– it is the mapping

$$manager_id \longrightarrow manager\ location.$$

For routing to work at all each manager must at least know the address of the manager responsible for the hash space before and after it. For routing to work efficiently a manager should know the locations of other managers throughout the hash space. A possible system with five managers is shown in **Figure 4.2**. The arrows in the centre show managers that know the location of another manager, each segment represents how the hash space and is divided up between each manager and the rings show the different types of information stored in the DHT.

### 4.2.1 Redundancy in the Network

The previously discussed technique of symmetric replication is used to provide a configurable level of redundancy, $R$, for all insertions. **Listing** 4.3 and **Listing** 4.4 shows the act of generating multiple keys and then storing them using recursive routing for $R = 2$.

**Listing 4.3:** Pseudocode for `put(value)`.

```
1 key1 = hash(value)
2 key1, key2 = symmetric_replication(key1, 2)
3 store(key1, x)
4 store(key2, x)
```

**Listing 4.4:** Pseudocode for `store(key, value)`.

```
1 manager = responsible(key)
2 if manager == this_manager
3   store(key, value)
4 else
5   tell manager to store(key, value)
```

To maintain the redundancy in the system, managers will query the two neighbouring managers in the network overlay at an interval of time $t$ to check they are still active. If a manager finds its neighbour has disconnected it will send a message to the manager responsible for the hash space symmetric to the missing manager's to request their key/values are re-added. 2 is only good value for $R$ if there is a very small chance of two nodes disconnecting within time $t$. If we have an estimate for the reliability of a node we can achieve a given reliability for the system (such as 0.9999%) by varying $R, t$. In the file sharing systems studied in [48], choosing a larger $t$ and $R$ will minimise the overhead of a DHT because storing files is generally done infrequently compared to checking if nodes are alive. While we store much smaller pieces of data in our system than in [48] we may perform store actions much more frequently (e.g. any time an actor is created, destroyed or migrates) at a rate that is highly dependent on the current application running within Stage#. During testing of Stage# with a small to medium sized number of nodes on a reliable local network I found that $R = 2$ and $t = 10$ seconds were suitable values. For larger, more distributed, networks we would expect larger values of $R$ to be used which is why the values for $R, t$ are left configurable.

Erasure coding is another technique used to provide redundancy by splitting information and storing each part on multiple hosts. We are generally dealing with quantities of data less than the size of a packet so splitting it would not bring the benefits obtained in [48] for much larger files.

Retrievals work in a similar way to insertions and only one key is needed. If there is a problem than the next key can be used. Having more than one key gives a choice of what key to look up first. This brings the advantage of being able to choose the path of lowest latency, a technique called proximity neighbour selection. Latencies can be based on prior communications or be predicted using synthetic coordinates [22]. While this and many other techniques are possible to improve the performance of Stage#'s DHT we have opted for simplicity in our first iteration of development.

## 4.3 The Stores

This section provides an explanation of each of the stores the DHT provides. Each store associates a different type of key to a value; for example, the names store uses an actor's name (which is a string) as a key and an actor_id as the value. Inserting key-value pairs into a store can be done asynchronously so can be considered a very cheap operation. Similarly, futures may also allow look-ups to be done asynchronously. If the result of the look-up is needed immediately the execution will block.

### 4.3.1 Names

Names are defined by the mapping

$$name \longrightarrow actor_id..$$

This is a non-injective function since, while an actor can have many names, at any given time a name maps to only one actor.

Actors can be associated to a name in two ways:

1. By any other actor that knows their actor-id using the primitive `name(name, actor)`. This includes self-naming when the actor_id is not specified.
2. At creation of the actor. This is used to implement singleton actors as explored in section 5.1.1

Here associating a name to an actor (a process we call *naming*) is synonymous to giving a Universal Actor Name to an actor in SALSA using a naming server or naming an object in Java using an Object Registry. It allows the location of an actor to be found at runtime and thus abstracts away the need for the programmer to know the location of the actor at write-time. The key difference with this and the other distributed systems mentioned is that there is no reliance on a naming server, instead it is the system itself that remembers the names.

Naming provides an easy way to register services for other actors to use. For example the print queue of **Listing 4.5** could be on any theatre, on any node and the word processor of **Listing 4.6** will still be able to find it.

**Listing 4.5:** PrintQueue actor.

```
1 class PrintQueue(MobileActor):
2   def birth(self):
3     name("Queue")
4
5   def add(self, document):
6     ...
```

We explain the execution of lines 4-5 of the `WordProccessor` actor in **Listing 4.6** in context of a system with 8 nodes as shown in **Figure 4.7**. First the `WordProcessor` actor asks its manager to find the actor with the name `'Queue'` (**1**). The request gets recursively routed by managers on different nodes (**2**, **3**) until node 2 is reached. The manager at node 2 know it is responsible for the hash space which contains `hash('Queue')` so queries its name store to find the actor_id for actor named `'Queue'` (**4**) and returns the result to the theatre that initiated the query (**5**). Now the actor at node

**Listing 4.6:** WordProccessor actor.

```
class WordProcessor(MobileActor):
  ...
  def print(self, text):
    queue = find_name("Queue")
    queue.add(text)
```

1 can send messages directly to the `PrintQueue` actor (**6**) and by storing the `PrintQueue`'s actor_id locally it will not need to perform a lookup next time it needs to send `PrintQueue` a message.

### 4.3.2 Types

Every actor that is created is added automatically to the type store. The type store is defined by the mapping

$$type \longrightarrow (list\ of\ actor_ids,\ list\ of\ types)$$

where every actor in the *list of actors* are all of the queried type and the *list of types* are the immediate subtypes of *type*, all subtypes can be found by recursion along the list of subtypes.

Storing subtypes can be useful. Imagine an actor needed to sort a list and there were actors of type `Heapsort` and `Quicksort`, both subclasses of `Sort`, in the system. This actor hierarchy obeys Liskov Substitution Principle so they are indistinguishable from each other and can always be substituted for one another [40]. Thus, our actor can use either `Heapsort` or `Quicksort` when an actor of type `Sort` is needed for identical results.

However, Stage# is not capable of checking or enforcing the Liskov Substitution Principle. This means it may not always be correct for Stage# to return a subtype of the type requested. For example, declaring a `Circle` to be a subtype of `Ellipse` is incorrect as the method `setSize(x,y)` will produced different results on each; they are distinguishable. The programmer must actively decide to use `find_subtype(t)` which will return an actor of type `t` or subtype of `t` or `find_type(t)` which does not find subtypes.

Finding actors of a specified type is needed because, unlike naming, it allows an actor to find one of possibly *many* actors in the system. First consider the definition `Logger` actor as given in **Listing 4.8**. The `Logger` writes data by sending it to persistent storage (i.e. a file on a disk). Now if there is an actor that represented a bank account and it had the requirement that all transactions were written to a file it could use the `Logger` actor to do this. The bank account actor does not care which `Logger` actor writes the transaction, as long as it is written *somewhere*. Since `find_type('Logger')` randomly returns a `Logger` in the system it also serves to distribute work among `Logger` actors. The `BankAccount` actor in **Listing 4.9** uses `find_type(type)` to achieve its requirements.

In general we find an actor by name if the actor with that name has state that we care about. If any actor of a given type can perform the actions we need and we do not require it to have a certain state then we can find the actor by type. Note that if Stage# strictly followed the Actor model and actors did not have state but replaced themselves with a different behaviour, i.e. type of actor, then we would not need names as types would be sufficient to distinguish different actors.

We also provide the primitive `find_all_types(type)` that will return all actors of a given type. This operation is no more expensive than `find_type(type)` since all types will hash to the same

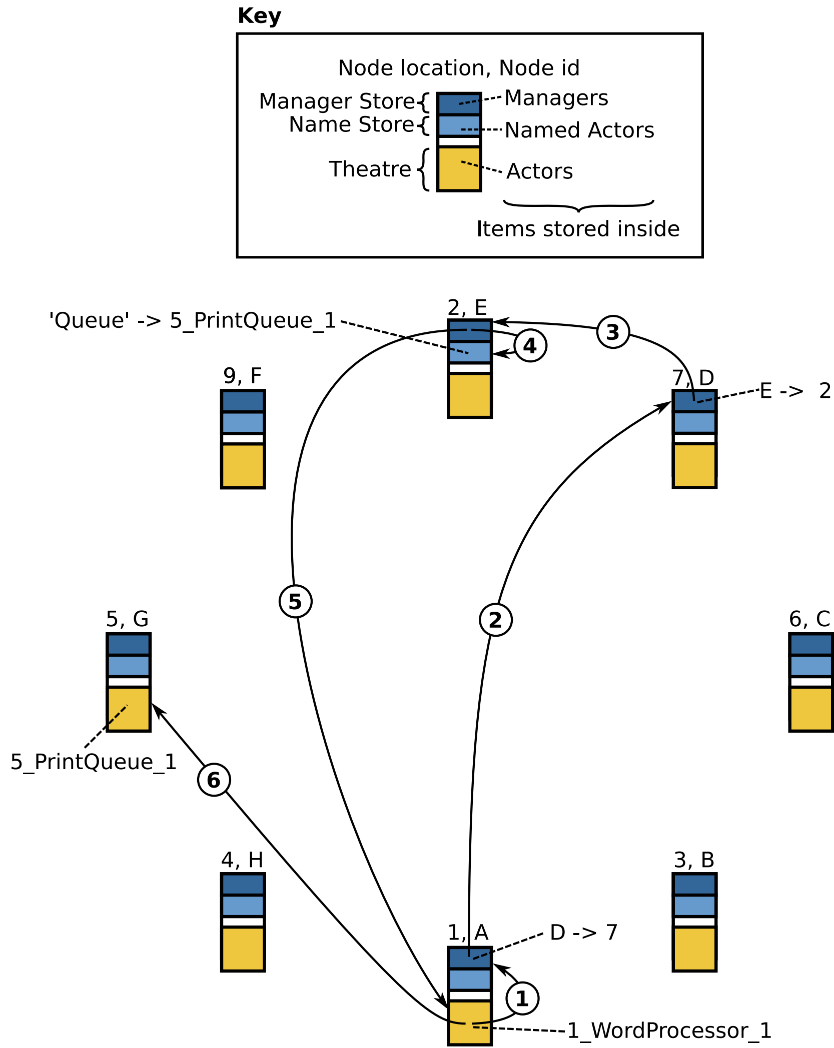



**Figure 4.7:** The execution of the `WordProcessor` actor in **Listing** 4.5. To simplify the diagram we only show the parts of the overlay and items in the stores that are used in this example.

**Listing 4.8:** Logger actor.

```
1  class Logger(MobileActor):
2    ...
3    def transaction(self, status, to, from, amount):
4      self.file.write(status, to, from, amount)
```

**Listing 4.9:** BankAccount actor.

```
1  class BankAccount(MobileActor):
2    ...
3    def transaction(self, to, amount):
4      log = find_type('Logger')
5          from = self.account_num
6      log.transaction('start', to, from, amount)
7      self.bank.move_money(to, from, amount)
8      log.transaction('complete', to, from, amount)
9    ...
```

key so will be stored at the same node (though more bandwidth will be used to return a greater number of actor_ids). In section 5.2 we discuss how this low level operation is useful in building the higher level operations for more advanced distribution schemes and for dynamic code replacement.

While randomly selecting a type to return will evenly distribute queries in the system it does not take into account the heterogeneity of the network; some theatres may be capable of dealing with more requests or have lower latency. `find_all_types(type)` allows the programmer to extend the system by writing their own selection function based on the requirements of the system.

Note that by keeping this type information on every actor, which changes every time an actor is created or destroyed, we are beginning to go beyond what a broadcast or centralised registry system would be capable of without being overloaded by traffic.

### 4.3.3 Migrations

The original Stage allows an actor to migrate from one theatre to another. After a migration a new actor_id is assigned that reflects the change in location. This breaks the promise we made before that knowing an actor_id is a sufficient condition to sending an actor a message– the old actor_id no longer tells us where to route the message and is not possible to query the old theatre as it may no longer exist (this is a likely case as closing a theatre is one possible trigger of migrations). To maintain our promise we introduce the migration store, which is defined by the mapping

$$actor_id \longrightarrow actor_id$$

and is used to find the new actor_id from the old one. Every time a migration occurs this store is updated automatically by the theatre receiving the actor. To prevent long chains of actor_ids needing to be traversed actors remember their set of previous actor_ids and update all of them in the migration store to map to the current id.

The migration store is used within the message sending function. A simplified version of the message sending algorithm is shown in **Listing** 4.10; if the theatre a message is being sent to is not reachable or the recipient actor is not at that theatre then the migration store is queried. A change in actor will cause the message sending function to notify the sender of the recipients new actor_id.

**Listing 4.10:** Consideration of migrations during message sending.

```
new_actor = null
for i in 0..retries:
  try:
    result = send(actor_id, message)
    if new_actor is null:
      return result
    else:
      return result, new_actor
  except HostNotReachable or ActorNotAtHost:
    new_actor_id = find_migration(actor_id)
    if new_actor_id not null:
      actor_id = new_actor_id
```

### 4.3.4 Source

The Source store could be considered the 'odd one out' of all the stores. While all the others stores are used to find running actors the source store is used to find the sources for individual actors. It is defined by the mapping

$$type \longrightarrow actor\ source.$$

Recall that Stage (being based on Python) only compiles actors into bytecode for the virtual machine at runtime, consequently it is the source of an actor we must know to create an instance of that actor.

The idea is for the source store to keep track of all sources in the system. When new actors need to be added at runtime then they should be added through the source store. The source store is used to:

- Prevent name clashes of types. If every actor is stored an error can be thrown if a source is added that is already named in the system. Name clashes risk two different actors being listed as the same type in the type store so should be avoided. This is used as an alternative to namespaces– we write

  `find_type('PrintQueue')` rather than `find_type('IO.Print.PrintQueue')`.

  We argue that namespaces on their own will not prevent name clashes at runtime because source could be dynamically loaded that uses the same namespace as another. The error thrown will make the programmer decide to take one of three possible actions; the actors are the same so there is no need to re-add the source, the actors are different so the source should have different names or one source is a newer version of the other so dynamic code replacement should be used instead.

- If a theatre needs to create an actor of a certain type but does not have the source locally it can go the distributed source store to find it. This situation is possible because Stage, being based on Python, is dynamically typed– there is no check at load time that all sources required exist locally.
- Decentralise retrieving sources and checking for updated sources throughout. If creating new actors relied on accessing a centralised file server we create a single point of failure in the system.
- Speed up migrations. Migrating an actor between nodes may also require source to be transferred, but this takes time, especially if the source is stored on disk and many sources need to be transferred. We would like to avoid this situation because migrations are often caused by shutdowns which are time-constrained. By storing the source in the store before a migration is triggered, only the state of the actor needs to be copied between hosts at the time of migration. The theatre receiving the actor can then query the distributed source store at leisure, even if the initiator of the migration is no longer part of the network.

## 4.4 Summary

Stage# uses a Distributed Hash Table (DHT) to store information about actors executing on nodes in the network. We store names as a means of locating a specific actor with state, we store type information automatically upon actor creation as a means of locating actors that provide a certain service, we store migrations so actors can be found even if they have moved hosts and we store sources so Stage# is not reliant on a separate file system and has some control on what actors are added to a running system. In the next chapter we build upon these stores to provide additional useful services to the Stage# programmer.

# Chapter 5

# Higher Level Concepts

Now we describe how the cheap and reliable stores, introduced in the previous chapter, can be used to build new abstractions for the programmer and better implement existing Stage# behaviour. To do this we:

- Modify the existing actor hierarchy of Stage# to clarify the abilities of the different base actor types.
- Explore higher level abstractions that use the cheap actor location mechanism to solve problems common in distributed systems such as the distribution of work and dynamic code replacement.
- Ensure our method for location, as well as Stage# itself, is suitable for all types of nodes in a distributed system. We consider low powered mobile devices and multicore machines.
- Provide the Visualiser tool that uses information from the distributed stores to create an image of the system that can be used to monitor or debug Stage# applications.

## 5.1 Actor Ontology

The original Stage had a handful of 'base' actor types that could be extended. For example the `NetworkSingletonActor` type could be extended to provide an actor that only allowed one instance of its type in the system and by extending `MobileActor` the programmer signalled to Stage the actor could be migrated to another host. Now we give reasons for one removal and one addition to these base actors.

### 5.1.1 Singleton Actors

Creating a local singleton or network singleton actor in the original Stage required inheriting `SingletonActor` or `NetworkSingletonActor` respectively. We remove this ability and instead provide the means to make singleton actors through naming.

The use of naming to create a network singleton is show in **Figure 5.1**. Line 6 shows passing a name to the constructor of ChatRoom. If a ChatRoom actor named room_name already exists a reference to that actor will be returned and if does not exist a ChatRoom actor will be created.

**Listing 5.1:** Creating singleton chat rooms.

```
1 class ChatUser(MobileActor):
2   def birth(self):
3     ...
4
5   def join_room(self, room_name):
6     room = ChatRoom(room_name)
7     room.join(self)
```

The power of using naming to create singletons is that it allows code reuse by giving different names to the same type– for example we could name a ChatRoom with 'stage-discussion' and then any future ChatRoom actors created with the same name will produce a reference to the same actor, but we could also create a ChatRoom named 'erlang-discussion'. This will also be a singleton in the sense that there can only ever be one 'erlang-discussion' chat room but it will be the same type of actor as named by 'stage-discussion'. We can force a singleton for a type by encapsulating the creation of the type in another actor– for example make the constructor of ChatRoom() return RealChatRoom('realchatroom').

To ensure that only one instance is ever created we need to lock the name store during the inserting a singleton value, this is a finely granular operation as we can lock just the name we are inserting. Recall that we must perform multiple insertions into the DHT to ensure redundancy; we use a method to decide which manager must perform the locking (implemented by choosing the manager responsible of the numerically smallest hash space) so we only need to lock one manager. The remaining insertion operations will continue for other managers only if the first one succeeds.

In the original stage SingletonActors were actors which only allowed one instance of that type per theatre. We can emulate this behaviour by appending the name of theatre to the name of the actor. For example, if we wanted to only allow one chat room per theatre we could create the singleton thus: ChatRoom(loc(self.actor_id) + 'ChatRoom') (where loc(self.actor_id) returns the theatre that the executing actor belongs to). We can similarly provide a means of having only one instance of a given type per node which was not possible on the original Stage.

### 5.1.2 Creatable Actors

The original Stage has a useful distinction between two types of actor; ordinary actors and mobile actors. Mobile actors inherited MobileActor which told the Stage virtual machine that it could move that actor between hosts if needed– it signalled to Stage that it had control over the location of the actor.

Now we introduce 'creatable' actors as a way to signal to Stage# that it can automatically create instances of that actor if it needs to. An actor is made creatable by inheriting CreatableActor. **Figure 5.2** shows the clear differences between the three base actors in Stage#, each specialisation of Actor giving more control to the Stage# virtual machine.

| | CreatableActor | MobileActor | Actor |
|---|---|---|---|
| Auto-created | X | | |
| Auto-placed | X | X | |

**Figure 5.2:** Creatable actors compared to existing actor base classes

Just as Stage placed conditions on the behaviour of a `MobileActor` (i.e. it cannot use location-specific resources) we introduce two conditions `CreatableActors`:

1. *Creatability*. Creatable actors must have a zero-parameter constructor so Stage# can create an instance of one.
2. *Interchangeability*. They must be stateless or only hold state that does not change behavior. This is to make instances of creatable actors interchangeable so if Stage# is asked to find a given creatable actor it can return any creatable actor of the same type to produce the same behaviour.

Creatability and Interchangeability allow Stage# to:

- Automatically create instances of creatable actors when needed without explicit construction. This is akin to invoking static methods in object oriented programming but the difference is that Stage# must create an instance of an actor in its own thread before sending it a message.
- Only have one instance of a creatable actor per theatre due to interchangeability. This means we can decrease the number of threads theatres need.

In section 5.4 we show how properties of creatable actors can be used to distribute work in multicore architectures.

Creatable actors are allowed to have non-behavior changing state, this is useful because it lets them cache variables such as the location of an actor they communicate with. We could also use state to store metrics about the actor such as the number of messages received. This type of information then could be sent to a metric collecting actor at regular intervals. If only one theatre is considered, a creatable actor has similarities to a static object– both will have a state shared across all instances on that node. However, because Stage# is distributed across many nodes (with non-shared memory) we cannot guarantee that two instances of creatable actors will always have the same state in a system of multiple theatres (but you could of course implement such behaviour using singleton actors).

## 5.2 Dynamic Code Replacement

Dynamic Code Replacement[1] is the act of updating code in the system at runtime with minimal disruption. It can be used to fix bug or update features on systems that cannot easily be restarted.

[1]Dynamic Code Replacement is often informally referred to as 'monkey patching', 'guerrilla/gorilla patching' or, the more violent, 'duck punching'.

We use update_method(actor, method_ref) to update the method on just one actor, where method_ref is the name of the function to be replaced on the remote actor and also the name of the local method that replaces it. The theatre this is called from, packages up the method and sends it to the remote actor as a priority message– the remote actor will immediately update its code after processing its current message. **Listing** 5.3 shows an actor that prints 'Hello' and **Listing** 5.4 uses the update_method() method to make the previous actor print 'HELLO!!!' instead.

**Listing 5.3:** Shout actor.

```
1 class Shout(MobileActor):
2
3   def birth(self):
4     self.message = 'Hello'
5     name('shouter')
6
7   def say(self):
8     print(self.message)
```

**Listing 5.4:** An actor that replaces code in one Shout actor

```
1 class MakeShoutLouder(MobileActor):
2   ...
3   def patch(self):
4     shout = find_name('shouter')
5     update_method(shout, say)
6
7 def say(self):
8   print(self.message.upper() + '!!!')
```

Normally we find what we actually want is to update code for all instances of a given type of actor and all future instances– if one actor has a bug then all actors of that type will have the same bug. We can extend MakeShoutLouder, as in **Listing** 5.5, to first update the source store so all future instances are correct and then iterate through updating all existing actors of the type. If we wished to we could also use the type store to find instances of subclasses and update them too, under the condition that they have not overridden the inherited method.

## 5.3 Scaling in all directions

We would like Stage# to easily 'scale' across multiple hosts. This means that, in the context of multicore machines, Stage# should be able to uses the resources of all cores without requiring any extra effort from the programmer or adversely affecting the performance of the system. In the context of mobile devices, we wish actors to be able to run locally on hardware of this type without modification.

**Listing 5.5:** An actor that replaces code in all actors of type `Shout`

```
class MakeAllShoutsLouder(MobileActor):
  ...
  def patch(self, module_name, source):
    store_source(module_name source)
    shouts = find_all_types('Shout')
    for shout in shouts:
      update_method(shout, say)

def say(self):
  print(self.message.upper() + '!!!')
```

### 5.3.1 Multicore

So far we have shown how we have helped Stage# scale across many hosts, now we explain the model we adapt to let Stage# fully use all cores of a single host. In the old Stage only one actor could run at once (multitasking) no matter how many cores the computer had[2], we would like true concurrency across cores since our actors do not share memory (beyond using services from same theatre) so there is no restriction of them running concurrently.

The new multicore model is shown in **Figure 5.6**; theatre processes are spawned for every core on the machine and share the same manager process. Originally the model omitted the shared information store but we found that when automatic balancing was added between cores that migrations were very frequent between theatres on the same host, this had two disadvantages:

- Every time a migration occurred the global migration, type and possibly name stores would have to be updated creating a large amount network overhead traffic (since every host with multiple cores would produce more traffic).
- The high frequency of migrations meant it was possible for actors to 'outpace' another trying to send it a message; consider the following sequence of events:

```
 1. A migrates from T1 to T2
 2. B sends T1 a message for A
 3. T1 receives message and replies with ActorNotFound
 4. B queries migration store
 5. B sends T2 a message for A
 6. A migrates from T2 to T1
 7. T2 receives message and replies with ActorNotFound
 8. B queries migration store
 9. A migrates from T1 to T2
10. GOTO 2
```

  Since `A` can migrate from `T1` to `T2` in less time than it take for `B` to attempt a message send, receive the response, query the message store and then resend the message it could take `B`

[2]This was not a problem due to Stage but with the underlying CPython Interpreter. Many interpreters, including most Python and Ruby implementations, have a global lock that prevents the need for locking to stop two threads accessing the same memory location. While this sounds bad it is an alternative to object-level locking which would be quite wasteful in Stage since objects are not shared between actors (beside a few elements of the theatre itself).

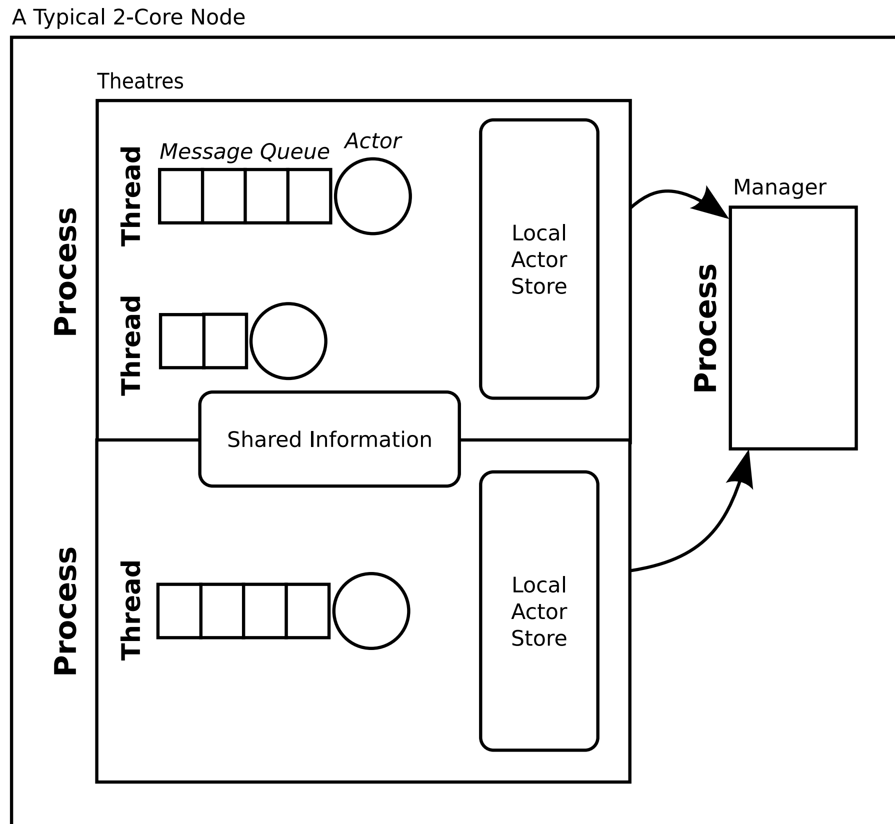



**Figure 5.6:** The number of theatre processes increases appropriately to fit the number of cores.

many attempts before the message is successfully sent. If the send is successful we have wasted a large amount of network traffic sending a, possibly large, message many times to the same host.

Our solution is to keep some shared information between all theatres processes on the same host. This shared information store is defined by the mapping:

$$identifier \longrightarrow theatre$$

where *identifier* a unique number that identifies actors on a host (for this we use the identifier that is already part of an actor_id) and *theatre* identifies the process of the theatre in which the actor is contained. This share is also a convenient place to keep information about the host such as a list of all the theatres running on the same host.

The new routing procedure for when a message reaches a theatre is given in **Listing 5.7**. Now if a message arrives at the correct host it is the receiving theatre's responsibility to ensure it arrives at the correct theatre while if it arrives at the wrong host it is the sender's responsibility– the migration store must be checked.

Our multicore model uses finely grained locking on the table of shared information since each theatre only ever needs to modify one entry at a time on creation, deletion or migration of an actor. This fine grained locking means that the shared information will not become a bottleneck in the system

**Listing 5.7:** How a message is routed when arriving at a theatre

```
1 if actor_id in local actor store:
2   add message to local actor's message store
3 else if actor_id in shared information:
4   update actor_id to refer to correct theatre
5   forward (actor_id, message) to theatre
6 else:
7   send ActorNotFound back to originator
```

so will scale from the 1-4 cores currently found in desktop PCs to many more.

### 5.3.2 Mobile Devices

The new theatre-manager structure of Stage# should be a legitimate model for all types of computers. We have made modifications to Stage# to provide a model that works on multicore machines, now we must meet the challenge of 'scaling' to mobile devices. We now explain how we produced an undiluted version of Stage# that works on PyS60[3], a Python interpreter for Symbian-based Nokia smartphones. Now we discuss the two main challenges we faced during porting Stage# to mobile devices[4].

#### Underpowered Theatres

The first challenge was coping with the much more modest specifications in terms of processor speeds and amount of RAM– for example, the current generation of high end smartphones have 300-500 MHz ARM processors and ∼128 MB of RAM. Similarly, when compared to wired networks, wireless connections have greater latency, less bandwidth and higher error rates, all of which vary in an unpredictable manner [62].

The first step was to change the way of thinking about the use of mobile devices in Stage#. PCs acting as nodes in Stage# are considered to be only semi-transient; they join the network, run actors for a period of hours/days, and then leave the network. While joined they give a share of their processing power to the Stage# system; they hold part of the global stores and they may receive actors from other nodes due to distribution or load balancing. In comparison, mobile devices need to be much more conservative in the processing power they share with the network so it is not feasible for them to play the same part as nodes. It is likely that they join the network in order to offload processing onto other nodes or to run an actor that is primarily focused on communication rather than computation, for example a chat client would be suitable for a mobile device, but a server that distributes the chats and handles a larger amount of traffic would not be. We simplify this statement in a rule to aid the Stage# programmer to remember good practices:

**Rule 1:** A mobile theatre should not perform computation on behalf of remote nodes.

[3] The PyS60 Project– https://garage.maemo.org/projects/pys60/

[4] Not discussed in depth here are the implementation challenges specific to porting between different operating systems and versions of the python interpreter. As a rule all operating system calls had to be re-written (such as file and socket access) especially if both operating systems *claimed* to follow the same standard API.

While this has to be enforced by the programmer, Stage# does provide some restrictions on its higher functions. For example, in section 5.4 we describe a method for distributing work across many nodes in a semi-automated manner using a contract net. Theatres on mobile devices will never offer to provide services to other nodes in this situation, thus **Rule 1** is enforced.

#### Transientness

The second problem is that mobile devices are much more transient than PCs in terms of connection to a network. They may move out of range of a wireless communication signal or could even run out of battery. Also mobile operating systems manage processes differently to PCs as a consequence of processor and memory constrains; background processes (i.e. processes not currently visible) are liable to be paused, closed or even not allowed by the operating system.

The new features of Stage# already help programmers to cope with the 'transientness' of mobile devices. Migrations enable actors that need to be kept alive to move from the phone onto a more reliable node when the theatre on the phone has to be closed. The name store allows re-location of actors on mobile devices if the device re-joins the network with a different IP address.

**Rule 2:** Actors running in a mobile theatre should make full use of registrations, lookups and migrations.

In the section 6.4 of the evaluation we show that by following **Rule 2** we can construct an application that continues executing actors started on a mobile device even if the device leaves the network. First the device creates and runs a named actor. When the device leaves the network it migrates the actor onto another node so the actor can continue executing and receiving messages– other actors can still send it messages because of the migration store. When the device re-joins the network it can find its actor by name and then migrate it back to itself.

As a result of higher communication latency and the previously mentioned problem of running background processes, managers are not created on mobile devices. Instead all mobile theatres are linked to a manager running on a standard node, as illustrated in **Figure 5.8**. Because theatres are associated with managers, but managers are not linked to theatres, it is easy for a theatre to change a manager during execution, this allows mobile theatres to choose another manager if the one they are currently using has a high latency or becomes unavailable.

## 5.4 Distribution of Work

We would like to make it easy for the programmer to parallelise work to complete a task in the shortest time possible. In Stage# actors are single threaded and only one actor per theatre can execute at once so to parallelise a problem we must send parts of the computation to actors in different theatres, possibly on different hosts. Offloading computation from one host to others in network may not always be beneficial if doing the work locally will take less time than the communication delay and the overhead in communication. For example, processing using the MapReduce model can have a high amount of start up overhead to distribute the information that needs to be processed among nodes [23]. If the computation cost is small compared to the amount of data needed to be processed and the benefits of parallelism across the network are diminished.

We aim for our method to:

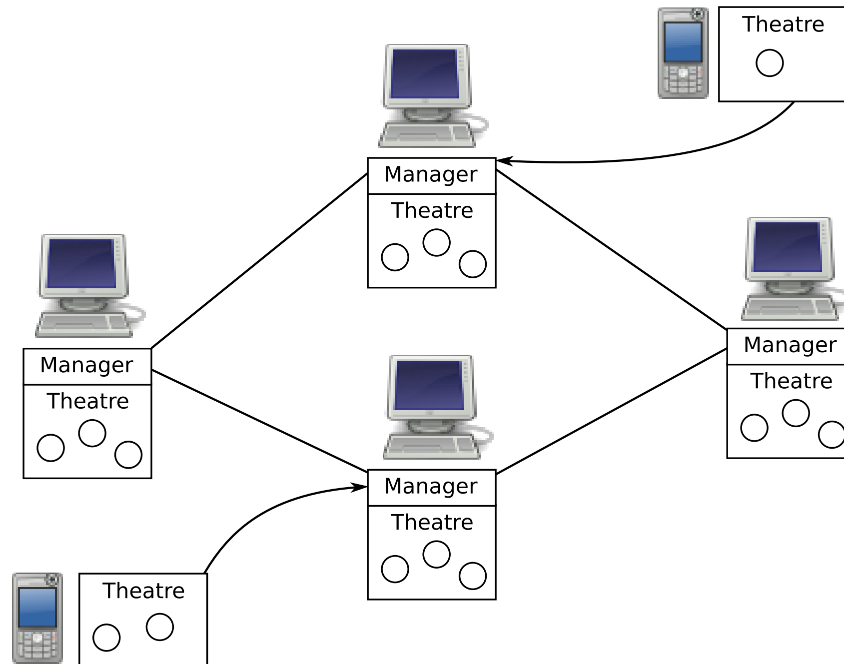


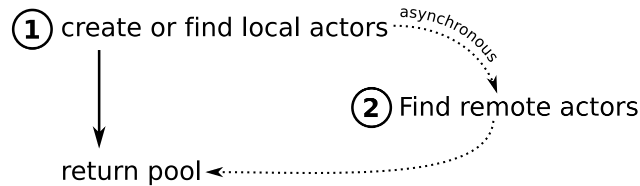

**Figure 5.8:** Mobiles devices do not run managers themselves, instead they use an existing one running on a PC or server.

- Avoid initial overhead that could increase the time needed for short computations
- Spread the work so it finishes in the shortest time possible
- Send work according to the capabilities of the receiving theatre; do not send work if doing so would slow the total computation time
- Remove from the programmer the responsibility of finding and creation of remote actors
- Avoid creating actors if existing ones of the required type can be re-used
- Parallelise fully across multiple cores.

### 5.4.1 Actor Pools

To help distribute work in Stage# we introduce the concept of actor pools. Actor pools contain the actor_ids of multiple actors of the same type and keep distinct the set of actors that are local to them and the set of actors that are on a separate host (and so have a higher communication overhead). A pool is created by `create_pool(type)` as shown in **Figure** 5.9. By searching the local actor store all local actors of the correct type are added to the pool and then the pool is returned. Occurring in parallel to this is an asynchronous call to the distributed type store using `find_all_types(type)` (introduced in section 4.3.2). This means the pool can be returned very quickly and be used immediately since it does not block while the remote actors are found.

If there are no local actors but the actor needed inherits `CreatableActor` (as described in section 5.1.2) then instances of the actor will be created automatically across all the theatres in the




1. In the actor's thread the local actor store is queried for actors of type `t`. The result is returned immediately.
2. In a separate thread remote actors of type `t` are found using `find_all_types(t)`. The pool is updated when the type lookup returns but invocations on the pool do not block while this is happening.

**Figure 5.9:** The actions taken when `create_pool(t)` is invoked.

host. Conversely, if there are no local actors but the actor does not inherit `creatable`, the actor pool will still return immediately but any invocations on it will block until remote actors are found.

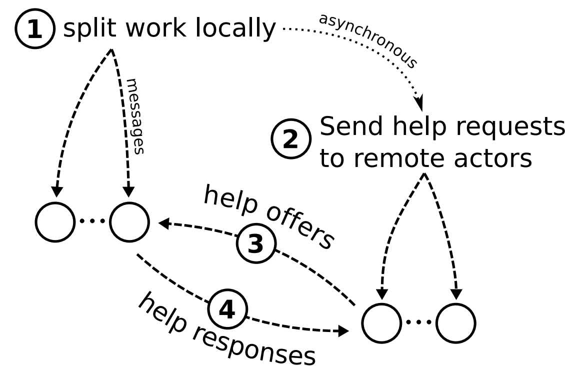
### 5.4.2 Pool Invocation

Now we have an actor pool we need to parallelise methods invoked on it. We combine the parallelisation technique of Parallel Maps and base our distribution negotiation technique on Contract Nets. Parallel maps are parallel versions of the Map higher order function and feature in many functional languages– such as the often reproduced `pmap` pattern in Erlang [11] and the `parMap` function in the Glasgow Haskell Compiler [41]. Our method of creating a pool has similarities to `pmap`, as written by Joe Armstrong, since both can spawn items (either actor threads or Erlang processes) automatically for every item in the list the map upon. The difference is that `pmap` expects a stateless function but in Stage# all functions are attached to actors which are not stateless, so we have to restrict ourselves to only spawning `creatable` actors. The result is that a method invoked on the pool would have the same result as a parallel map would, but we hope to compute all the results quicker than a parallel map that only parallelises across cores by offloading work onto remote actors using a contract net.

**Figure 5.10** explains what happens when a method is invoked on a pool. First all the work is split between local actors and they immediately begin processing messages. In parallel to this *help request* messages are sent to remote actors in the pool. The remote actors will respond to this request with a *help offer* if they are in a theatre which is below a certain load and have a message queue shorter than the actor requesting help. This help offer will be put to the front of the message queue.

A help response can then be sent, containing a subset of the workload (messages) from the actor's message queue. The number of messages to send is a function of the number of help offers received, the number of messages in the local actor's queue and the number of messages originally divided up between the actors on that host. The actor would have already processed some messages in queue so can estimate its average message processing time, if this multiplied by the number of messages in its queue is less than the round trip time between itself and the actor that offered help (which the actor also knows due to the help request-offer exchange) then there is no benefit from distribution so the actor ignores the help offer.

An example of an actor pool used to return all the prime numbers in a given list is shown in **Listing 5.11**, note that lines `4-5` could be replaced with




1. All the messages generated from the map are split between all the local actors in the pool.
2. In a separate thread the remote actors are asked for help.
3. The remote actors offer help if they have low load and short message queues (when compared to the actor requesting help).
4. The local actors accept the help if it is beneficial for them to do so based on the rate they process messages, the latency between them & the actors offering help and the size of their message queue. The amount of work they send is calculated from the number of help requests they receive and the size of their message queue.

**Figure 5.10:** The actions taken when `pool.method([args])` is invoked.

```
checker = PrimeChecker()
results = map(checker.isPrime, numbers)
```

for the same result, but with no parallelisation between cores or nodes.

**Listing 5.11:** A method that returns a list of all numbers in the original list that are prime using actor pools. Assumes the assumption of a `PrimeChecker` actor with a method `isPrime(x)` that returns `True` iff `x` is prime.

```
1  class Primes(MobileActor):
2
3    def prime_subset(numbers):
4      checker_pool = create_pool('PrimeChecker')
5      results = checker_pool.isPrime(numbers)
6      return [numbers[i] for i in range(len(results)) if results[i]]
```

## 5.5 Visualising and Debugging Stage#

In Stage# we often describe the many nodes, theatres and managers that combine at runtime as ‘the system’, this is quite an abstract term that does not help with understanding the specifics of the runtime state such as:

- What nodes form the system?

- What addresses do these nodes have?
- On what ports are theatres and managers running on?
- Is the node a PC or mobile device?
- What actors exist in a theatre?
- How is information in the stores distributed?

If the Visualiser component is enabled then certain actions in Stage# (such as creating a theatre, creating an actor or modifying a store) create an asynchronous notification that is sent to the Visualiser. The Visualiser processes this information and can generate an image[5] with the components of the system laid out in a logical manner. An example of what is produced by the Visualiser is shown in **Figure 5.12**.

The Visualiser was originally created to help us, as the maintainers of Stage#, to check managers were correctly and evenly distributing information across the stores. Continuous testing of Stage# has shown that, in its current state, that inserting and retrieving information from the distributed store of Stage# is reliable so there is less value of using Visualiser for this purpose than there was during the development of Stage#. This would change if the features of Stage were once again being extended.

The Visualiser continues to be useful for any programmer using Stage#. We created a web-based interface from which a continually updated image of the system is shown. The interface also provides a 'Debugging Mode' to allow the programmer to step through execution at their own pace. This is achieved by changing the communication to the Visualiser from asynchronous notifications to blocking synchronous calls. Thus when debug mode is enabled actors will pause when interacting with the Visualiser at important points of execution; creation of actors, creation of theatres & managers and the modification of the name or type stores.

## 5.6 Summary

Many of the ideas introduced in this chapter use the stores constructed in **Chapter 4**; for example, the method given for dynamic code replacement in section 5.2 relied on both the type and source stores to ensure all actors in the system were updated. In fact the usefulness of the ideas presented in this chapter *depend* on the correctness, functionality and speed of the stores. In our evaluation of Stage we explorer this further, using the performance of these higher level abstractions to as a means of evaluating our use of a Distributed Hash Table.

[5]The image produced is in the Scalable Vector Graphics (SVG) format, which has the advantage of being relatively easy to produce and manipulate in a language good at text processing (i.e. Python) without the need to manipulate pixels. SVGs are also easily resizable and can be embedded in web pages.

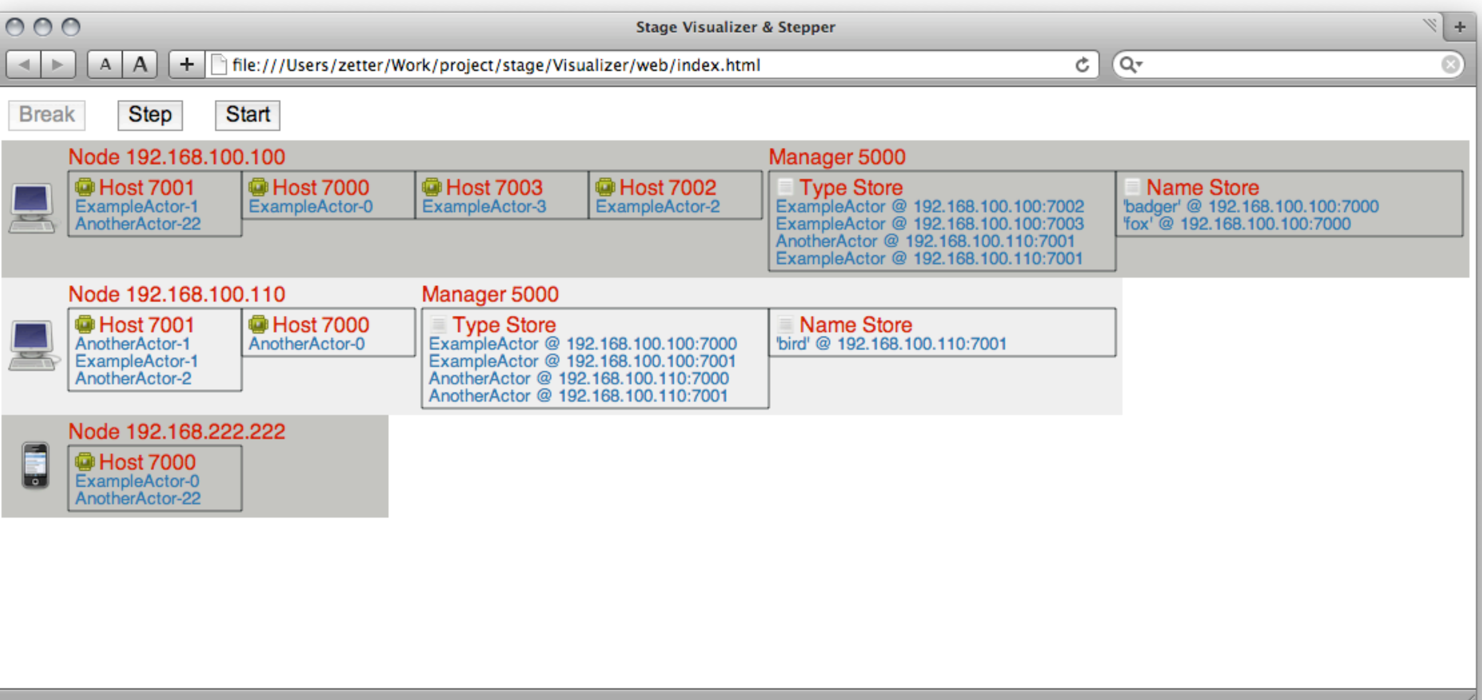



**Figure 5.12:** The Visualiser is accessible through a web based interface.

# Chapter 6

# Evaluation

> "Locating actors has always been the weakest part of [the original] Stage" [15]
>
> *John Ayres, Stage Creator*

The original Stage's main goal was to provide an implementation of a language that supported concurrency. 'Actor Discovery' was listed as a feature of the language, it would be impossible for actors to communicate in a dynamically distributed language without some method of discovery, but only provided a simple method for doing so (described as 'hints') with the aim that it could be extended. We used this as an opportunity to explore our novel use of a Distributed Hash Table (DHT) to provide actor location and have had success in solving Stage's deficiency. We evaluate the DHT-based lookup system and our other features of Stage# by considering:

- If the DHT-based lookup system we have constructed is scalable when compared to other process location mechanisms.
- If, from the programmers perspective, we have made the best implementation choices and provided the most useful features for taking advantage of the DHT. To answer this question we shall look at a language with similar goals as Stage# and compare the clarity and conciseness of the languages as well as features of the two languages.
- If the new features of Stage# aid the programming for multicore PCs and mobile devices within distributed systems. We shall produce two separate applications that use the new features of Stage# and make comparisons between programming them in the new and the original Stage.

## 6.1 Scalability of Process Location Mechanisms

Now we compare the characteristics of using a DHT for process location with the two prevalent methods:

- Broadcast, as found in Erlang [2]. Every node in the network remembers every item of information. For example, when a new node joins the network it must broadcast its location to every other node in the network. Similarly for nodes leaving the network and naming of processes.

- Server with broadcast, as found in Jini [1] and easily constructible in a broadcast system. Broadcasting is used to inform nodes of the existence of servers in the network. If we add/remove servers we use broadcasting to notify all the nodes. Multiple servers are allowed to provide redundancy, each having a copy of the same data. It would be unfair to consider servers without broadcasting abilities– new servers would not be able to be added to the network without reliance of another, possibly centralised, lookup system. For example the use of DNS in SALSA seen in section 3.3.

All three methods are considered for a distributed system across the Internet; multicasting cannot be used and there is non-uniform latency between hosts

### 6.1.1 Cost of storing information

How much bandwidth is used to store information? In the broadcast method every insertion causes communication to every node in the network. There are two problems with this; a lot of network traffic is generated and an insert operation will take longer to perform as the number of nodes in the network increases because an ever increasing number of nodes need to be notified every insertion. In contrast, both the server and DHT methods produce traffic relative to the redundancy level of the system so as more nodes are added the same number of insertions occur.

#### Exponential Growth

We wanted to try to prove the claim in [52] that storing information in Erlang (and broadcast systems in general) created unmanageable amounts of network traffic and show that Stage# did not suffer from the same problem.

We used a test program to simulate process registration and lookups in a distributed system that runs on $N$ nodes. For every node in the system the program ran on it would:

- Create 10 Erlang Processes/Stage# actors and register each one with the global naming service.
- Perform $N$ process lookups (one for every node in the system) and send each process that it finds a message.

We run this program on varying numbers of nodes and measured the total traffic in the system. The full method, source for the test programs and results are given in **Appendix A**. A summary of the finding are shown in **Figure 6.1**, this graph shows clearly the exponential growth in traffic in the Erlang based system; the traffic increases by a factor greater than 2 every time two more nodes are added. As expected, Stage# shows a gradual growth in traffic. Stage# is even disadvantaged in this experiment because it is automatically storing type information of the created actors which Erlang is not. We would expect similar a graphs if we compared traffic from adding and removing nodes in the system rather than processes because Erlang broadcasts these events to every node in the system too. This is a bad characteristic to have for Erlang– we would expect adding more nodes and processing power into a system to improve performance but instead nodes and routers become saturated with traffic, producing an outcome opposite to the desired one.

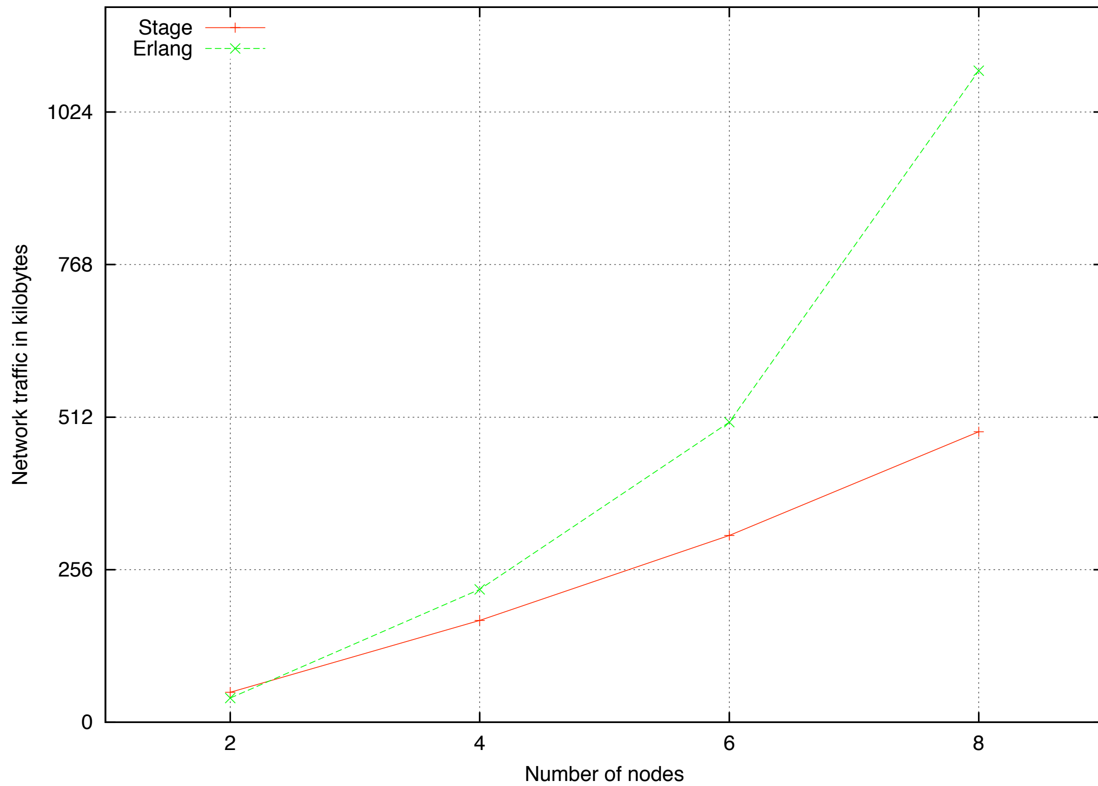



**Figure 6.1:** Traffic generated to register 10 processes in Erlang and Stage#

### 6.1.2 Lookup hops

How many routers and nodes must a lookup request go through before it reaches the node that can reply? This allows us to compare how long a lookup will take between the different methods. Lookups have no cost in a broadcast system (because 'every node knows everything') and this is the greatest benefit of broadcasting. In both server and DHT based systems a lookup has to be routed through the network. When using servers the request travels directly to the correct place. If there is a choice of multiple servers the closest one can be used. Our DHT method works in a similar way but is routed across an overlay network. Each step in the overlay network may require many hops on the actual network so lookups will not be expected to be as fast as using the server method. However, DHT does guarantee a lower bound for hops on the overlay network and there is a multitude of research on optimising DHTs, for example on how to construct the optimal overlay that is closest to the underlying network as to provide the most direct routing [64] or on how to pick the best route when multiple choices exist [22]. We avoid further discussion of the optimisations possible– we are more concerned with scalability, i.e. the question 'Do hops increase if the network grows?'. The answer is yes but at a slower and acceptable rate; DHTs have been proven scalable [54]. Also note that futures in Stage# mean lookups will be performed asynchronously so will not always block, there would be a similar benefit in a server based system that used futures.

### 6.1.3 Cost of adding nodes

How much traffic is generated when a node is added or removed? There is no overlay network in the broadcast method, instead every node keeps track of every other so to insert a new node every existing node in the system must be contacted. Just like the insertion operations we saw before, this also does not scale to large numbers of nodes. When using servers new nodes only have to be registered with all the servers. If the node is itself to be a server then all nodes need to be contacted. In the DHT method the addition of a node generates a small amount of traffic. At a minimum the two nodes adjacent to the new node in the overlay must be contacted, in practice we also contact other nodes in the overlay to build up routing tables. This is slightly more communication than is needed using the server method since it will be routed by the overlay, but is still scalable.

### 6.1.4 Redundancy

There is no configurable redundancy in the broadcast model but, because there are so many copies of everything, we find the same problem as we would in a highly redundant server or DHT model– consistency. Storing information across the network takes a long time and there are many more nodes to inform so there is a greater chance that two nodes hold conflicting information. The server model provides redundancy if we have multiple servers, each known to nodes in the network. The DHT uses a redundancy level as defined in a global variable. If a server fails or leaves the network then all the information stored in the failed server needs to be copied to another server to maintain redundancy, this may require manually adding a new server to the network. In our DHT method each manager only stores a subset of all the information in the network so a smaller amount of data needs to be copied when a failure occurs. Redundancy is maintained automatically by re-adding information stored on the lost nodes to other nodes; unlike the server model there is no need to add (or designate existing nodes as) servers.

### 6.1.5 Planning

The server model suffers from a high level of complexity in the planning stage. Imagine you want to create a distributed application (such as a web service) that needed to the processing power of many machines. You would have to think and answer each of the following questions:

- How many servers do we need so each one is not overwhelmed with requests? This depends on how frequently information is stored and retrieved. In Stage# requests are distributed across the system– one node never has the chance to be 'overwhelmed'.
- What machines should be servers? We may consider the latency of the server relative to other nodes in the network
- How do actors/objects become aware if servers change? The standard approach is the use of broadcasting but this will not scale for large numbers of nodes. Instead we could partition the network but this leads to more planning– how should it be partitioned? Does the architecture of the system allow it to be partitioned easily?
- If a server fails what should replace it to maintain redundancy?– We may need standby servers for this.

| Model | Broadcast | Server with broadcast | DHT |
|---|---|---|---|
| Insertion/removal cost | high | low | low |
| Lookup hops | zero | low (over network) | medium (over overlay) |
| Add/remove node cost | high | low (unless server) | low-medium |
| Level of redundancy | fixed | configurable | configurable |
| Amount of planning | none | high | none |
| Bottleneck | network capacity | server capacity | none (distributed) |

**Figure 6.2:** Comparison of the three different location methods.

- If the network grows will the number of servers be adequate? Perhaps new servers will need to be added manually by administrators. This is not the case for Stage#– location services provided by managers grow with the number of nodes in the network.

### 6.1.6 Conclusion

We summarise the properties of the three methods in **Figure 6.2**. We wish to explain why a DHT is the only one that provides a solution for process location in a truly distributed system.

Elements of the broadcasting method preclude it from scaling; every node has to remember all information stored as well as all other nodes in the system. We have shown how traffic increases rapidly when new nodes are added to the system. Ultimately the bandwidth of the network and the ability of nodes to handle so many store requests becomes the limiting performance factor. For this reason broadcasting is only feasible for small systems and a network infrastructure capable of handling the generated traffic. It also serves time-critical systems well because of the instant lookup time.

Creating a large server based system has been shown to be less than straightforward. By having a separation between servers and ordinary nodes we have to manage the scaling of servers separate to the scaling of nodes even though they are very much linked; more nodes likely mean more processes which in turn means more insertions and lookups. Another problem with the server model is that all data is stored at every server which means that even if multiple servers are used the same amount of insertions occur at each one, possibly too many for it to handle.

Neither broadcasting or the use of servers sit well in a truly decentralised system. Due to a restriction of the number of nodes based on network capacity broadcasting is better suited for local networks rather than the internet. The use of servers entails a centralised authority to plan and manage them. Stage# with its DHT exhibits most of the characteristics of a server based system, but surpasses the server model by scaling without planning actions at runtime. Also, unlike the server model, insertions are distributed across many nodes affording us to store much more information. In Stage# we take advantage of this by storing types, migrations, and sources along with providing a naming service. In fact the stage is set for storing whatever we would like in a cheap and distributed manner with its own level of redundancy. We conclude that using a DHT for the location of actors/objects, as demonstrated in Stage#, is the preferred option over the server and broadcast models for *any* large (i.e. not confined to a LAN) distributed system.

## 6.2 The language

### 6.2.1 Comparison to JXTA

So far we have avoided giving examples of JXTA due to its verboseness. However, since it is one of only a handful of languages that use a DHT to distribute information about processes we may gain insight into Stage#'s choice of syntax and semantics by comparing the two.

#### The JXTA Client and Server

We take a client-server example from a set of JXTA tutorials provided by Sun's JXTA development website[1]. The full code is **250 lines in length** and is provided in **Appendix C**.

When running this system the server registers the service it provides and then waits to receive messages. The client finds this registered service and then sends a string to the service for it to be printed.

#### The Stage# Client and Server

**Listing 6.3:** ServiceServer actor.

```
1  from Actors.keywords import *
2
3  class ServiceServer(MobileActor):
4
5    def birth(self):
6      name("server")
7
8    def print_message(self, msg):
9      print("Received message: " + msg)
```

**Listing 6.4:** ServiceClient actor.

```
1  from Actors.keywords import *
2
3  class ServiceClient(MobileActor):
4
5    def birth(self):
6      server = find_name("server")
7      server.print_message("Hello my friend!")
```

A Stage# program with the same purpose as the JXTA one is shown in **Listing 6.3** and **Listing 6.4**. Note that the client could be condensed to the one-line:

```
find_name("server").print_message("Hello my friend!")
```

[1] JXTA development website - https://jxta.dev.java.net/

JXTA and Stage# are both written on top of high-level languages, both run in a virtual machine, both share the same goal of enabling people to write distributed applications, both make a point of including mobile devices in their definition of a distributed system, both have switched to a DHT-based network overlay in their second version and both are one of the very small set of languages that store information about the running system in a DHT. So why does the source for the JXTA and the Stage# programs look so different? We focus on the source for the JXTA client-server program to see if it is performing important actions that Stage# is inadequate to provide.

#### Abstractions

JXTA has explicit abstractions for every part of the network. For example, communication is not just routed over the physical network as in Stage# but instead goes through the configurable abstraction of *pipes*. Pipes can be made to follow the overlay network rather than the physical, the advantage of this is that we know by construction that the overlay network is guaranteed to be traversable whereas the physical route chosen may not work (firewalls could stop traffic). We argue that the decision to route messages on the overlay should be done automatically at run time– the programmer does not know when writing the program if this problem will occur as it is dependant on where the application is deployed.

#### Security

JXTA organises nodes into *peer groups*. When JXTA used the broadcasting technique this was necessary to stop large amounts of traffic being generated by restricting broadcasts within a group. Now peer groups are less important from a technical standpoint due to the use of a DHT but are used to organise the network. Different peer groups can have different security policies (i.e. requirements for joining) and provide separate namespaces for advertisements. Stage# does not provide security features; while it will be trivial to restrict joining the network (by a password) and make traffic between nodes confidential (by encryption) once a node has joined the network it is trusted completely. In Stage# untrusted nodes should not become part of the network and instead may only access network services through a limited interface. Nodes with different degrees of trust can be managed in JXTA by placing them in different peer groups. Each peer group has its own security requirements to join and has restrictions on what services can be used from within the peer group.

#### What is Stored

While Stage# stores simple key-value pairs in its DHT, JXTA stores *advertisements* for the same purpose. Advertisements have much more information; including a name, description, creator and specification (Server excerpt, lines 93-99):

```
93  ModuleSpecAdvertisement mdadv = (ModuleSpecAdvertisement)
94          AdvertisementFactory.newAdvertisement(ModuleSpecAdvertisement.getAdvertisementType());
95  mdadv.setName("JXTASPEC:JXTA-EX1");
96  mdadv.setVersion("Version 1.0");
97  mdadv.setCreator("sun.com");
98  mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
99              mdadv.setSpecURI("http://www.jxta.org/Ex1");
```

However when the client looks for advertisements (lines 59-74) none of this information is actually used beside the name. JXTA allows different items to be advertised; *peers, peergroups, pipes, services, content* and *endpoints* [44]. This is quite a complex list of possibilities that the programmer has to understand. Stage# stores 'advertisements' for actors. Nodes and sources are also stored but these are handled automatically by the virtual machine so do not have the opportunity to confuse the programmer. Since actors are programmable entities we can use them as interfaces for many of the JXTA abstractions; for example, we could create actors which encapsulate *content* and then name the actor with an identifier for that content.

### Message Passing

Everything that is sent to another actor is implicitly a message in Stage# (and in the original Stage). Messages in JXTA have to be explicitly created before they are sent (client excerpt, lines 87-91):

```
87  String data = "Hello my friend!";
88  Message msg = new Message();
89  StringMessageElement sme = new StringMessageElement("DataTag", data, null);
90  msg.addMessageElement(null, sme);
91  outputPipe.send(msg);
```

This certainly makes distributed programming awkward since this has to be done for every remote call. JXTA retains the imperative nature of Java making message receiving equally awkward– see the '`while (true)`' loop that the server (line 116-147) uses to process messages. Forcing the programmer to build their own basic control structures such as message receiving leads to complicated code and a higher probability of bugs. Creating concurrent threads for message handling will require use of the Java threading library.

### Exception Handling

The JXTA code is full of exception handling. We have omitted handling exceptions in our Stage# example; we could check that a name is returned when lookup occurs but we do not have to as exceptions will just cause the current message being processed to fail rather than be propagated up the call stack as done in Java. We believe a Stage# is lacking in a well defined exception model that will allow exceptions to be propagated back to the originator of the message. This may be useful feature to have in later versions of Stage# (discussed in section 7.3). Stage#'s virtual machine will retry many functions automatically a fixed amount of times, such as resending messages if a failure occurs or re-performing a lookup (on a replicated hash) if the first attempt fails, avoiding the try-catch block inside the '`while (true)`' loop in the client (line 59-74) which repeats the lookup action forever until it succeeds.

### Conclusion

The designers of JXTA have made even the simple task of sending a string across the network 250 lines of code, forcing the programmer to write out common operations such as creating a message. The Stage# virtual machine tries to do as much work as possible in order to make network communication identical to local communication. Both Stage# and JXTA provide distributed stores but the complicated act of inserting and looking up information in these stores in JXTA does not reflect

the cheap and quick operation that it is. This, I believe, dissuades programmers from taking full advantage of the stores. However, the JXTA code has highlighted an area of Stage# that could be improved; exception handling, and reminds us that Stage# lacks a security model.

## 6.3 Distribution Mechanism

This section goes through the steps needed to adapt an existing program to use an actor pool and then evaluates if actor pools do indeed fulfil their aims (given in section 5.4) based on the running of the program.

### 6.3.1 Adapting the Code

To test the ability of actor pools to parallelise work I modified the trapezoid benchmark from the original Stage. Previously work was naively divided between known theatres on the system after manually specifying which theatres to use, now we use an actor pool;

```
worker_pool = get_pool(TrapWorker)
results = worker_pool.approximate(samples, lowers, uppers)
result = sum(results)
```

I found that I needed to additionally modify some code of the `TrapWorker` actor for it to work with the map, previously the `TrapWorker` actor was created and then sent a message containing the part of the function it was responsible for evaluating, this information was saved as state and then computation began when a `start` message was received. Now the `TrapWorker` actors have had their state removed (i.e. they are now functional) so they can be used in a map.

### 6.3.2 Results

I ran the trapezoid approximation on five machines, each with a dual core processor. The results are plotted in **Figure 6.5** and full details of the test and results are in **Appendix B**. The graph shows the load of each core in the system. The trapezoid application is started on node 1 at about +1.5 seconds and immediately work is distributed to both the local cores on the machine. At +6 seconds the actors on node 1, after receiving all responses to the request for help, redistribute some of their work causing the load to increase on the other nodes. Then from +23 seconds the load for some nodes starts dropping. The results are collected by the actor on node 1 which prints the approximation at about +28 seconds.

Now we compare the results to the original aims of the distribution mechanism (section 5.4):

- Avoid initial overhead that could increase the time needed for short computations– work starts immediately on one node and is not distributed across the network until we know if it is beneficial to distribute work. This means we never make computation take longer than it would on a single host and avoid needlessly creating traffic on the network.
- Spread work so it finishes in the shortest time possible– this has been done to a point since this is in conflict with our first aim. We have accepted that full parallelisation should not occur until we know more information (the estimated time to finish, load of other nodes, etc.).

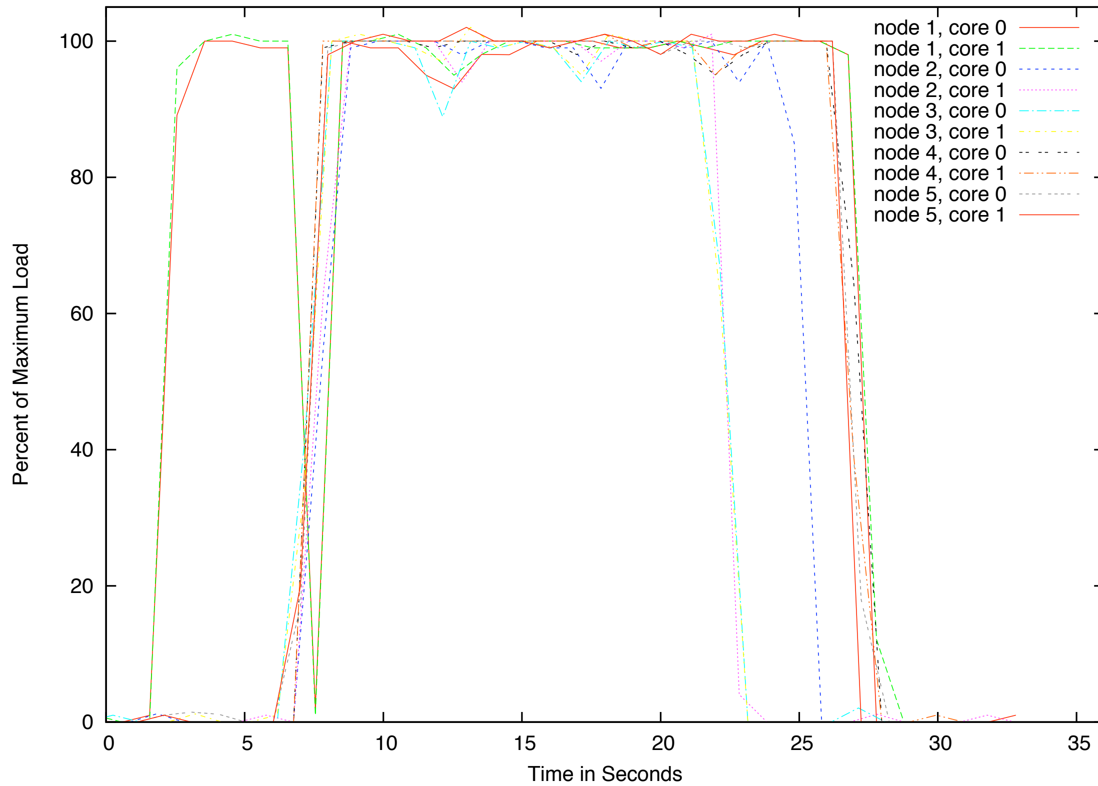



**Figure 6.5:** Distribution of trapezoidal approximation in Stage#

- Send work according to the capabilities of the receiving theatre; do not send work if doing so would slow the total computation time– we have achieved this by looking at the current load of the machine and use the number of cores of the machine to estimate its speed. We have found this to work well on a selection of heterogeneous machines (single, dual and quad core) in a local network for computationally bound work. Work that required different resources of the machine (such as space on the hard disk for processing a large amount of data) would require a different rating system.

- Remove from the programmer the responsibility of finding and creation of remote actors– this is successfully done by using the type store and overlay network.

- Avoid creating actors if existing ones of the required type can be re-used– we use the type store and creatable actors to use this. The type store is queried quickly compared to the time it takes to process one message so we do not have to 'wait' for its result.

- Parallelise fully across multiple cores– also achieved (see the pairs of cores for each machine in the graph) note that the redistribution of work is also parallelised between the cores of the node that created the pool.

### 6.3.3 Conclusion

Mapping work across an actor pool is an easy way for a programmer to reliably parallelise work in Stage#. It serves as an example for the implementation of other constructs that adapt existing parallelisation techniques for a ridged program, perhaps deployed on a grid-like system, to a dynamic peer to peer system; by taking into account the dynamic state of the system we can automatically decide the best way to distribute work. We have given an implementation for a map but there is no reason why the idea could not be extended to other higher order functions (such as a fold, filter, etc.).

There is not a comparable feature in the original Stage; if the programmer wanted to distribute work they would have to first find the remote theatres, then manually create actors in them and lastly manually divide the work between the remote actors, all without taking into account the capabilities and load of each theatre. Stage# does this all automatically.

By programming using constructs like actor pools applications can be made to automatically scale when needed; in this case just adding more nodes to the system will mean the work will be distributed across a greater number of actors. Not one line of code needs to be changed.

## 6.4 Yapper, A Distributed Micro-blogging Service

Yapper is constructed to provide similar features to the popular web service Twitter. Every person has an account. They can post messages to this account (called yips) and subscribe to the accounts of other Yapper users. A user's own messages and the messages of all the accounts they are subscribed to are posted to the user's wall, ordered by the date of posting. For example, a run of Yapper for a person with the account name 'Badger' that follows 'Bird' and 'Monkey' could create the wall:

```
+ At 09:31 I yapped: Hello everyone!
+ At 10:02 Bird yapped: Hi Badger! How are you?
+ At 10:04 I yapped: good, trying out this new micro-blogging tool
+ At 11:19 Monkey yapped: it even works on my phone!
```

Yapper comprises of many different accounts (representing people), an implementation of Yapper would need to find the location of accounts so messages can be sent to them. Accounts have to be indexed by a human-readable tag for users to be able to subscribe to them. There is also the challenge of being able to use Yapper from a mobile device. We also want to make Yapper scalable by not having any centralised bottlenecks in the system.

The idea for Yapper came from Cheeper, a similar application made to demonstrate the features of Thorn, a language in development by IBM. Thorn presents itself as a scripting language that focuses on concurrency and shares features with the Actor model and Erlang [63]. Yapper and Cheeper both have to deal with many concurrent conversations so provide a way to demonstrate the languages ability to handle concurrency.

We wish to show that Stage# provides clear features and abstractions for Yapper's requirements and also compare it to Thorn's implementation.

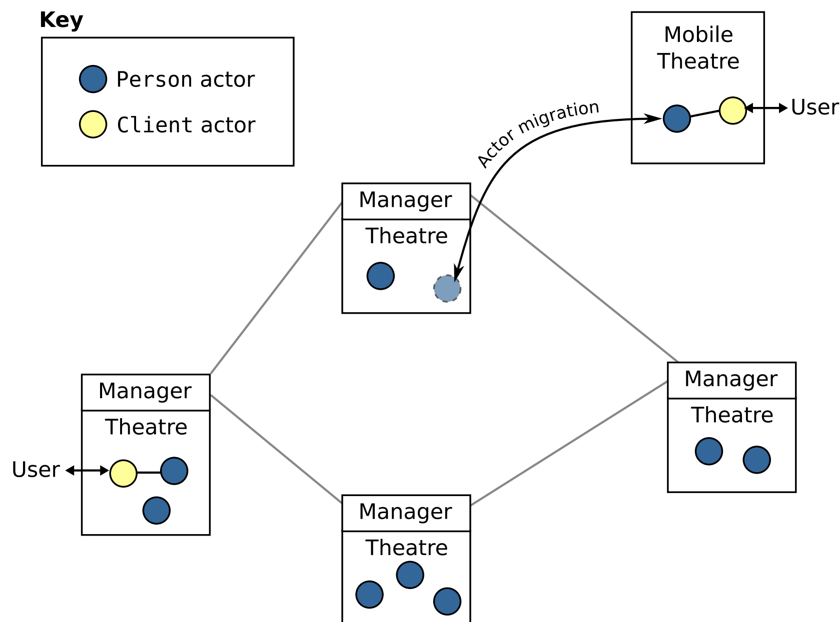



**Figure 6.6:** Yapper running across many nodes.

### 6.4.1 Implementation

We provide a Yapper implementation written in Stage#. It is given in **Appendix D**, is ∼100 lines in total and took two hours to write and test. We use two types of actors; `Person` actors which represent a Yapper account and `Client` actors which is an interface between users and accounts.

`Person` actors remember the state of an account; this includes the contents of the wall and a list of accounts being followed. They also know how to display themselves and do so differently based on where they are currently running, this allows flexibility for different devices; for example if the `Person` actor was running on a mobile device it could display a space-optimised interface that is different to the one it displays on a PC.

`Client` actors are run when a user wants to use Yapper. They log the user into the system and find the `Person` actor associated with their account. They handle user input, passing it to the `Person` actor.

We also have `Holder` actors which do no work themselves, instead they are created on theatres that are to 'hold' `Person` actors when not attached to a client. A more advanced implementation could adapt Holder actors to provide authentication or a persistent store.

### 6.4.2 Use of the Stage#'s Features

Many of Stage#'s features helped produce the account-centric implementation of Yapper. We leverage the distributed stores to do all the location and indexing of accounts for us instead of a centralised data store.

When `Client` actors find the `Person` actor that encapsulates the user's account they try and migrate

it to the current location. **Figure 6.6** shows the moving of a `Person` actor when a mobile device joins the system. Migration of the actor means that any work that actor has to perform is carried out by the node the actor has migrated too. This means Yapper does not have to provision large amounts of centralised processing power to deal with user initiated events. If a node has to leave the system `Person` actors can be migrated to another node so their state is saved. `Client` actors do not hold important state so are destroyed. A suitable node is found for the `Person` actor to migrate to by using `find_type("Holder")`, since `find_type` will return a random type in the system this will distribute `Person` actors across many nodes.

Our `Person` actors are actually named singletons (see section 5.1.1) which concisely models that we never have more than one account with the same name. We are allowed to move actors about so freely because we can always find them again by using our distributed store. Because actors are expected to move between hosts we do not even cache any of their locations in our implementation, instead choosing to use the name store each time.

We also tested our method of dynamic code replacement (section 5.2) while Yapper was running. We changed the implementation of the method called when of a user writes a new message to perform additional validation on user input– a check on message length. The change was applied successfully to all `Person` actors with no noticeable negative impact on the running system.

### 6.4.3 A Scalable Architecture

A possible implementation of Yapper would be to use a centralised store. This is what Cheeper (a similar application written in Thorn) does; a single server process guards access to a centralised database [18]. This has clear drawbacks– the server will be limited to the speed of the node it is located on and if the speed is not adequate than more servers will need to be used. Of course the Thorn program could be adapted to use a decentralised database and distribute work across multiple servers but doing so is not an easy task. Even more difficult is the ability to do so at runtime since at current thorn does not have a mechanism for process location; the address of the server is given to the client when it starts. Contrast this to Stage# where the address of any node in the system can be used to join the network, and the specific address of the server used is not needed.

By encapsulating the state of the system in actors and registering the location of these actors in our distributed store we avoid the use of any centralised data store. Computation occurs not on a server but at the current location of the actors, this would not be possible in Thorn as there is no means to locate clients on the edges of the System; all communication must go through the server. If we wanted to scale our implementation of Yapper we would just need to add new nodes to the system. This can be done dynamically with no change of code needed causing the distributed store to immediately be re-distributed and over time the `Person` actors will migrate to this machine.

### 6.4.4 Conclusion

In our implementation of Yapper the distributed store allows Stage# to abstract away inter-host communication allowing us to send actors messages whichever node they are running on. We take advantage of this by migrating actors to clients at the ‘edges’ of the network and performing computation there.

Yapper is just one example of an application Stage# can create but we can see that we could use the same mechanisms when building any scalable application, in particular Stage# makes it easy to:

- Push Actors out to the edges of the system. Here they can still be found and sent messages because of the name and migration stores.
- Perform client-client communication, avoiding having to use centralised servers.
- Distribute all lookup and registrations across the system by using the stores.
- Distribute data and computation across the system, possibly dynamically using naming to find resources at runtime.

We could re-use these in any distributed application. If needed to, Stage# can still create centralised actors on a given host and, because of the distributed store, even centralised actors can migrate if needed to; for example, the node they are running on leaves the system.

## 6.5 Summary

We have shown the advantages of using a distributed hash table over broadcasting or a server based model are due to better scalability and no requirement for planning. We then compared Stage# to JXTA concluding that, while both have the same goal, Stage# provides similar features with fewer and clearer abstractions (with the exception of security and exception handling– discussed further in section 7.3). We then showed that actor pools met their original aims and provide an easy way of distributing work that could not be done in the original Stage without hard-coding information about the system. Lastly we used Yapper to show how Stage# can be used to build scalable applications, listing the specific abilities of Stage# that could be used to make any application scalable.

# Chapter 7

# Conclusions and Further Work

First we discuss the use of a distributed hash table for process location as being applicable to any distributed language, then we look at the Stage# language.

## 7.1 Using a Distributed Hash Table for Process Location

We have shown in section 6.1 that there are key differences between DHTs and the existing process location models (i.e. those based on the use of broadcasting or centralised servers) and conclude that DHTs are the only model that provide a scalable method for storing process information in distributed systems. Whereas existing process location models are restrictive in what information can be stored because the network/resources can easily be overloaded, using a DHT allows us to store large amounts of data in a distributed and redundant manner. This has triggered us to change our thinking from what a process location service *needs to store* to what it would be *useful to store* to aid the programmer in creating distributed applications.

For an example of what a DHT allows us to do that exiting process location models do not, consider the type store in Stage#. The location of all actors created are automatically put in the type store and we can do this for every actor because storing information does not create large amounts of traffic (as it would if broadcast) or overload a centralised server. A collection of actors of a given type can cheaply be found since they all would be registered at the same node (due to a consistent hash function) and we take advantage of this property by querying the type store when distributing work. Compare this to the Java based JXTA which also uses a DHT to provide location services. JXTA's use of a DHT was added some time after the language was created and did not influenced the design of the language itself so does not provide useful features that exploit properties of its DHT as Stage# does.

Our Yapper implementation makes use of many of Stage#'s DHT-based features including the creation of singleton actors, dynamic code updating, the finding of remote actors by name and the guarantee of being able to find actors by name after migrations. We end up with a feature-rich Yapper client using very little code. This together with the other example applications we have created in Stage# demonstrate how our additions to Stage allow the creation of scalable, distributed applications that can exploit the concurrency of many different types of computers in a distributed system.

## 7.2 The Stage# Language and Applications

Stage# now works on mobile devices and performs better on multicore systems; any full application/prototype written in Stage# can easily take advantage of these two resources.

> "...Stage would be an excellent language for quickly prototyping or modelling distributed systems..." [14]
>
> *John Ayres, Stage Creator*

I agree with the above statement both in terms of distributed systems created using Stage# and if used to describe Stage# itself– Stage#'s high level Python implementation and relatively small code base means it is a great distributed language to experiment with (as I have done with DHTs). During development of Stage# and its applications I found my Visualiser tool (section 5.5) invaluable for understanding and debugging the current problem. The Visuliser strengthens Stage#'s position as a prototyping language by aiding development and providing better understanding of the runtime state of the applications prototyped. I would add to John's statement that, because Stage# now provides useful features that other distributed languages do not, in some situations Stage# may provide a far more elegant and scalable solution to a problem.

I believe that I have provided the correct abstractions to the programmer. After programming applications and tests in Stage# and comparing it to my own experience of programming across multiple computers (including using Erlang, Python, Java and AJAX) I found Stage# to be the most pleasant. I believe this is because in Stage# it takes very little to find actors on other hosts so one can concentrate on the task at hand rather than managing the network.

One domain where Stage# could be useful is providing web services. This is already an area that favours high-level languages such as Python. Producing scalable web pages is a hard task, but there are already generic solutions available. The current question is on how to write complex web based services, often with live interaction between users, which must also scale across many machines. The recently announced Google Wave Protocol[1] tries to tackle this problem, focusing on the interaction between many clients across many servers (possibly owned by different organisations). Stage# could solve the same problem in a different way, instead providing a decentralised client-orientated solution.

## 7.3 Further Work

Some ideas for future development of Stage# are discussed by John in [14] and [16]. While we have solved the problem of Actor location problem, and side-stepped the limitation of CPython's global interpreter lock by a new multicore model. Some of John's observations still stand, in my opinion the most important is security.

Stage# does not provide any security features. Although would be trivial to restrict joining the network (by a password) and make traffic between nodes confidential (by encryption), once a node has joined the network it is trusted completely meaning they could execute any code on any node in the network. One approach would be to follow the JXTA model as described in section 6.2.1– partition nodes into separate sub-networks and have different requirements for joining each sub-network. Another advantage that JXTA is that it can use Java's sandbox model to restrict the actions

[1] Google Wave– http://wave.google.com

that untrusted code can perform. CPython does not provide sandboxing[2] so implementing such a feature in Stage# would be a non-trivial task.

While we have done our best to test developing a variety of applications for Stage# we have been time-constrained. As a result we have avoided developing large applications in Stage# or deploying Stage# applications across very large numbers of nodes, perhaps using a networking testbed. While I believe doing so will provide further justification for Stage#'s model, I also think it may prompt refinements, or perhaps extensions, to be made to Stage#'s choice of stores.

Stage did not have performance as a main goal and nor does Stage# (besides the ability to scale). It is likely that Stage# will continue to perform poorly in distributed benchmarks, such as the thread ring, when compared to a high-performance distributed language such as Erlang. A big performance improvement could be had by changing from using a Python thread (which is an operating system thread) for every actor to instead using a more lightweight threading model[3] or a thread pool. Creating a thread pool has challenges since a naïve implementation runs the risk of deadlock– the threads must be able to stop executing an actor mid-method if it blocks. Stage#'s performance is heavily dependent on Python's so a recent push to provide a better performing Python interpreter, such as Google's Unladen Swallow project[4], may help in the long term.

Another area for improvement in Stage# that became clear in the comparison to JXTA was the lack of a well defined exception model. If an error occurs while one actor processes a message perhaps messages should be returned to the originating actor, or perhaps a `ErrorHandling` actor could exist at every theatre that had overridable behaviour for many common causes for failure (such as a missing parameter or incorrect type).

Finally, I think a good addition to Stage# would be a theatre that could run within a web browser. I believe it would be possible to use the same theatre model as Stage# has for mobile devices. The implementation would have to be Javascript-based and theatres could either interpret a subset of Python or a new a syntax could be constructed so actors could be written directly in Javascript. The actors in Javascript could communicate to other nodes using AJAX. One problem of a Javascript implementation is that the current generation of web browsers become unresponsive when Javascript is executing so a continually running entity such as an actor may not be appreciated by the user. Currently background Javascript tasks are supported by Google's browser add-on Google Gears[5] and are planed to be in a future release of Mozilla Firefox[6] .

## 7.4 Closing Remarks

Stage# shows how useful it is to have more advanced actor location features built into a language. It also shows that by using a distributed hash table we do not need to compromise the scalability of the system or limit what runtime information we choose to store. Stage# re-implements many of Stage's features, such as actor migration, to use the distributed store so programmers will gain immediate benefits from using Stage#. They can then benefit further by using the new features Stage# provides (such as actor pools, multicore theatres, support for mobile devices, and the Visualiser) to create scalable applications.

[2] The Python wiki provides an overview of (the lack of) sandboxing in Python– http://wiki.python.org/moin/How_can_I_run_an_untrusted_Python_script_safely_(i.e._Sandbox)

[3] Stackless is a Python interpreter that provides lightweight microthreads– http://www.stackless.com/

[4] Unladen Swallow project– http://code.google.com/p/unladen-swallow/

[5] Google WorkerPool API reference– http://code.google.com/apis/gears/api_workerpool.html

[6] Mozilla DOM worker reference– https://developer.mozilla.org/En/Using_DOM_workers

It is my hope that Stage# continues to evolve, providing means to test features for distributed languages, and maybe will influence the design of other distributed languages.

# Appendix A

# Storing Information

## A.1 Conditions

The programs were run on eight Linux virtual machines running under a VMware host. The total network traffic sent between them was logged using `tcpdump`.

## A.2 Test Programs

Both Erlang and Stage# programs follow a similar structure. When running either, the `server` is started with the number of nodes in the system and an identifier for the current node. The `server` then spawns 10 `worker` processes/actors on that node.

### A.2.1 Erlang Server

```
1  -module(server).
2  -import(worker, [ping_server/0]).
3  -import(lists, [seq/3, concat/1]).
4  -export([server/2, start/2]).
5
6  server(Name, Num_nodes) ->
7    receive
8      {start} ->
9        Ls = seq(0, 9, 1),
10       Names = [list_to_atom(concat([a , Name, "_", L])) || L <- Ls ],
11       Pids = [spawn(worker, start, [N]) || N <- Names],
12       server(Name, Num_nodes),
13
14       timer:sleep(15000),
15
```

```
16      Ns = seq(1, Num_nodes, 1),
17      Lookup_names = [list_to_atom(concat([a , N, "_", (L rem Num_nodes)])) || N <- Ns ],
18      Lookup_pids = [global:whereis_name(N) || N <- Lookup_names],
19      [P ! ping || P <- Lookup_pids]
20  end.
21
22  start(Name, Num_nodes) -> spawn(server, server, [Name, Num_nodes]),
```

### A.2.2 Erlang Worker

```
1  -module(worker).
2  -export([start/1]).
3
4  start(Name) ->
5    global:register_name(Name ,self()),
6    ping_server(Name).
7
8  ping_server(Name) ->
9    receive
10     Other ->
11       io:format("pinged"),
12       ping_server(Name)
13   end.
```

### A.2.3 Stage# Server

```
1  import time
2  import sys
3
4  from Actors.keywords import *
5  from Demo.Naming.worker import Worker
6
7  class Server(MobileActor):
8    def birth(self, my_name, num_nodes):
9      ls = range(0, 10)
10     store_names = [str(my_name) + "_" + str(l) for l in ls]
11     [name(n, Worker()) for n in store_names]
12
13     time.sleep(15)
14
15     ns = range(1, num_nodes + 1)
16     lookup_names = [str(n) + "_" + str(n % num_nodes) for n in ns]
17     actors = [find_name(n) for n in lookup_names]
18     [a.ping() for a in actors]
```

```
19
20 def start():
21   Server(int(sys.argv[1]), int(sys.argv[2]))
22
23 if __name__ == '__main__':
24   initialise(start)
```

### A.2.4 Stage# Worker

```
1 from Actors.keywords import *
2
3 class Worker(MobileActor):
4   def birth(self):
5     pass
6
7   def ping(self):
8     print("pinged")
```

## A.3 Results

These show the total amount of traffic logged in bytes. We use the arithmetic mean to calculate the average values.

### A.3.1 Erlang

| Nodes | Run 1 | Run 2 | Run 3 | mean |
|---|---|---|---|---|
| 2 | 42092 | 41056 | 41392 | 41513.3 |
| 4 | 228298 | 228040 | 228544 | 228294 |
| 6 | 516922 | 514680 | 515544 | 515715 |
| 8 | 1120529 | 1117473 | 1120571 | 1119524 |

### A.3.2 Stage#

| Nodes | Run 1 | Run 2 | Run 3 | mean |
|---|---|---|---|---|
| 2 | 53261 | 49896 | 50928 | 51361.7 |
| 4 | 175835 | 174974 | 174183 | 174997 |
| 6 | 321268 | 321942 | 319712 | 320974 |
| 8 | 503364 | 499282 | 494667 | 499104 |

# Appendix B

# Actor Pools Test

## B.1 Conditions

A total of trapezoids 30000000 were used to approximate the same function as the original Stage (see [14]) in the interval of 0-20.

I ran the approximation on five Linux machines; `edge{10, 11, 12, 13, 14}.doc.ic.ac.uk`. Each machine has a 2.13GHz Intel Core 2 Duo processor and 2GB of RAM. The test was run multiple times and there was only a small change in the allocation of work each time. We only display the results for one run and not averages since in doing we would lose the information of what machine is currently loaded and which is not.

The tables following show the load of each core the machine (i.e. percent of time each core is busy) taken by reading scheduled to occur every second. Note that the readings are not synchronised between nodes but all nodes do share a synchronised clock, the output of which is shown at every reading.

## B.2 Results

| edge10 | | | edge11 | | | edge12 | | |
|---|---|---|---|---|---|---|---|---|
| time | core 1 | core 2 | time | core 1 | core 2 | time | core 1 | core 2 |
| 0.58 | 0 | 0 | 0.85 | 0 | 0 | 0.25 | 1 | 0 |
| 1.57 | 0.99 | 0 | 1.84 | 1.22 | 0 | 1.24 | 0 | 0 |
| 2.55 | 89.11 | 96.04 | 2.82 | 0 | 0 | 2.22 | 0 | 0 |
| 3.55 | 100 | 100 | 3.8 | 0 | 0 | 3.21 | 0 | 1.19 |
| 4.56 | 100 | 101.01 | 4.79 | 0 | 0 | 4.19 | 0 | 0 |
| 5.56 | 99 | 100 | 5.77 | 0 | 1.02 | 5.19 | 0 | 0 |
| 6.56 | 99 | 100 | 6.76 | 0 | 0 | 6.17 | 0 | 1.04 |
| 7.55 | 2.02 | 1.01 | 7.84 | 55 | 63 | 7.16 | 41.41 | 34.34 |
| 8.53 | 100 | 100 | 8.82 | 99 | 100 | 8.14 | 100 | 100 |
| 9.53 | 99 | 100 | 9.83 | 100 | 100 | 9.15 | 100 | 101.01 |
| 10.54 | 99 | 101 | 10.83 | 100 | 100 | 10.15 | 100 | 100 |
| 11.54 | 95 | 99 | 11.83 | 100 | 100 | 11.14 | 99.01 | 99.01 |
| 12.54 | 93 | 95 | 12.84 | 98 | 94 | 12.14 | 88.89 | 96.97 |
| 13.55 | 98.04 | 98.04 | 13.84 | 100 | 100 | 13.15 | 100 | 102.02 |
| 14.54 | 98 | 100 | 14.86 | 100 | 100 | 14.15 | 99.01 | 99.01 |
| 15.54 | 100 | 100 | 15.85 | 99.01 | 100 | 15.14 | 100 | 100 |
| 16.55 | 100 | 100 | 16.85 | 99 | 100 | 16.14 | 99.01 | 99.01 |
| 17.54 | 100 | 99 | 17.85 | 93.07 | 97.03 | 17.13 | 94.06 | 95.05 |
| 18.54 | 99.01 | 99.01 | 18.85 | 100 | 100 | 18.14 | 100 | 101.01 |
| 19.59 | 99.01 | 99.01 | 19.84 | 100 | 100 | 19.15 | 99 | 100 |
| 20.65 | 100 | 100 | 20.84 | 99.01 | 99.01 | 20.15 | 100 | 100 |
| 21.65 | 99.01 | 99.01 | 21.84 | 100 | 101.01 | 21.15 | 99 | 100 |
| 22.64 | 97.98 | 100 | 22.83 | 93.94 | 4.04 | 22.15 | 66.67 | 62.63 |
| 23.64 | 100 | 100 | 23.84 | 100 | 0 | 23.15 | 0 | 0 |
| 24.73 | 100 | 100 | 24.82 | 85 | 0 | 24.13 | 0 | 0 |
| 25.73 | 100 | 100 | 25.81 | 0 | 0 | 25.16 | 0 | 0 |
| 26.76 | 98 | 98 | 26.82 | 0 | 0 | 26.15 | 0 | 0 |
| 27.79 | 0 | 12.12 | 27.8 | 0 | 1.16 | 27.13 | 2.06 | 0 |
| 28.77 | 0 | 0 | 28.79 | 0 | 0 | 28.12 | 0 | 0 |
| 29.84 | 0 | 0 | 29.81 | 0 | 0 | 29.1 | 0 | 0 |
| 30.83 | 0 | 0 | 30.8 | 0 | 0 | 30.09 | 0 | 0 |
| 31.81 | 0 | 0 | 31.78 | 0 | 1.01 | 31.09 | 0 | 0 |
| 32.82 | 1.01 | 0 | 32.77 | 0 | 0 | 32.08 | 0 | 0 |
| | | | 33.17 | 0 | 0 | | | |

**Results (continued)**

| edge13 | | | egde14 | | |
|---|---|---|---|---|---|
| time | core 1 | core 2 | time | core 1 | core 2 |
| 0.86 | 0 | 0 | 0.12 | 0 | 0 |
| 1.84 | 0 | 0 | 1.11 | 0 | 0 |
| 2.82 | 0 | 0 | 2.09 | 1 | 1 |
| 3.81 | 0 | 0 | 3.08 | 1.46 | 0 |
| 4.79 | 0 | 0 | 4.06 | 1.15 | 0 |
| 5.78 | 0 | 0 | 5.04 | 0 | 0 |
| 6.76 | 0 | 0 | 6.03 | 0 | 0 |
| 7.82 | 99.01 | 100 | 7.01 | 17 | 20 |
| 8.82 | 100 | 100 | 8 | 99 | 98 |
| 9.83 | 100 | 100 | 9 | 100 | 100 |
| 10.82 | 100 | 100 | 9.99 | 100 | 101.01 |
| 11.82 | 99 | 100 | 10.99 | 100 | 100 |
| 12.82 | 100 | 100 | 12 | 100 | 100 |
| 13.82 | 100 | 100 | 13 | 100 | 102.02 |
| 14.94 | 100 | 100 | 14 | 100 | 100 |
| 15.98 | 100 | 100 | 15 | 100 | 100 |
| 16.98 | 98.02 | 100 | 16 | 99.01 | 99.01 |
| 17.98 | 100 | 101 | 16.99 | 100 | 100 |
| 18.99 | 99.01 | 99.01 | 18 | 100 | 101.01 |
| 19.98 | 100 | 100 | 19 | 100 | 100 |
| 20.98 | 98 | 100 | 20 | 98.04 | 98.04 |
| 21.98 | 95.05 | 95.05 | 21.11 | 100 | 101 |
| 22.97 | 98.02 | 99.01 | 22.11 | 100 | 100 |
| 23.98 | 100 | 100 | 23.11 | 99 | 100 |
| 24.97 | 100 | 100 | 24.1 | 100 | 101.01 |
| 25.97 | 100 | 100 | 25.19 | 100 | 100 |
| 26.97 | 64.44 | 38.89 | 26.2 | 100 | 100 |
| 27.98 | 0 | 0 | 27.23 | 18 | 0 |
| 28.96 | 0 | 0 | 28.25 | 0 | 0 |
| 29.95 | 0 | 1 | 29.24 | 0 | 0 |
| 30.93 | 0 | 0 | 30.3 | 0 | 0 |
| 31.92 | 0 | 0 | 31.29 | 0 | 0 |
| 32.9 | 0 | 0 | 32.27 | 0 | 0 |
| 33.29 | 0 | 0 | | | |

# Appendix C

# JXTA Client and Server

Follows is a example of a JXTA client and server. The code is taken from [3] and is unchanged in function (it has been reformatted and comments have been removed).

## C.1 The JXTA Server

```
1  package tutorial.service;
2  import net.jxta.discovery.DiscoveryService;
3  import net.jxta.document.AdvertisementFactory;
4  import net.jxta.document.MimeMediaType;
5  import net.jxta.document.StructuredTextDocument;
6  import net.jxta.endpoint.Message;
7  import net.jxta.endpoint.MessageElement;
8  import net.jxta.id.IDFactory;
9  import net.jxta.peergroup.PeerGroup;
10 import net.jxta.pipe.InputPipe;
11 import net.jxta.pipe.PipeID;
12 import net.jxta.pipe.PipeService;
13 import net.jxta.platform.ModuleClassID;
14 import net.jxta.platform.NetworkManager;
15 import net.jxta.protocol.ModuleClassAdvertisement;
16 import net.jxta.protocol.ModuleSpecAdvertisement;
17 import net.jxta.protocol.PeerGroupAdvertisement;
18 import net.jxta.protocol.PipeAdvertisement;
19
20 import java.io.File;
21 import java.io.StringWriter;
22 import java.net.URI;
23 import java.net.URISyntaxException;
24
25 public class ServiceServer {
26
27     static PeerGroup netPeerGroup = null;
28     static PeerGroupAdvertisement groupAdvertisement = null;
29     private DiscoveryService discovery;
30     private PipeService pipeService;
31     private InputPipe serviceInputPipe;
32     private NetworkManager manager;
33     public final static String PIPEIDSTR =
34         "urn:jxta:uuid-9CCCDF5AD8154D3D87A391210404E59BE4B888209A2241A4A162A10916074A9504";
35
36     public static void main(String args[]) {
37         ServiceServer myapp = new ServiceServer();
38         System.out.println("Starting Service Peer ....");
39         myapp.startJxta();
```

```
40            System.out.println("Good Bye ....");
41            System.exit(0);
42        }
43
44        private void startJxta() {
45            try {
46                manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC, "ServiceServer",
47                        new File(new File(".cache"), "ServiceServer").toURI());
48                manager.startNetwork();
49            } catch (Exception e) {
50                e.printStackTrace();
51                System.exit(-1);
52            }
53
54            netPeerGroup = manager.getNetPeerGroup();
55            groupAdvertisement = netPeerGroup.getPeerGroupAdvertisement();
56            System.out.println("Getting DiscoveryService");
57            discovery = netPeerGroup.getDiscoveryService();
58            System.out.println("Getting PipeService");
59            pipeService = netPeerGroup.getPipeService();
60            startServer();
61        }
62
63        public static PipeAdvertisement createPipeAdvertisement() {
64            PipeID pipeID = null;
65            try {
66                pipeID = (PipeID) IDFactory.fromURI(new URI(PIPEIDSTR));
67            }
68            catch (URISyntaxException use) {
69                use.printStackTrace();
70            }
71            PipeAdvertisement advertisement = (PipeAdvertisement)
72                    AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType());
73
74            advertisement.setPipeID(pipeID);
75            advertisement.setType(PipeService.UnicastType);
76            advertisement.setName("Pipe tutorial");
77            return advertisement;
78        }
79
80        private void startServer() {
81
82            System.out.println("Start the ServiceServer daemon");
83            try {
84                ModuleClassAdvertisement mcadv = (ModuleClassAdvertisement)
85                        AdvertisementFactory.newAdvertisement(ModuleClassAdvertisement.getAdvertisementType());
86
87                mcadv.setName("JXTAMOD:JXTA-EX1");
88                mcadv.setDescription("Tutorial example to use JXTA module advertisement Framework");
89                ModuleClassID mcID = IDFactory.newModuleClassID();
90                mcadv.setModuleClassID(mcID);
91                discovery.publish(mcadv);
92                discovery.remotePublish(mcadv);
93                ModuleSpecAdvertisement mdadv = (ModuleSpecAdvertisement)
94                        AdvertisementFactory.newAdvertisement(ModuleSpecAdvertisement.getAdvertisementType());
95                mdadv.setName("JXTASPEC:JXTA-EX1");
96                mdadv.setVersion("Version 1.0");
97                mdadv.setCreator("sun.com");
98                mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
99                mdadv.setSpecURI("http://www.jxta.org/Ex1");
100               PipeAdvertisement pipeadv = createPipeAdvertisement();
101               mdadv.setPipeAdvertisement(pipeadv);
102               StructuredTextDocument doc = (StructuredTextDocument) mdadv.getDocument(MimeMediaType.XMLUTF8);
103               StringWriter out = new StringWriter();
104               doc.sendToWriter(out);
105               System.out.println(out.toString());
106               out.close();
107               discovery.publish(mdadv);
108               discovery.remotePublish(mdadv);
109               serviceInputPipe = pipeService.createInputPipe(pipeadv);
110           }
111           catch (Exception ex) {
112               ex.printStackTrace();
```

```
113             System.out.println("ServiceServer: Error publishing the module");
114         }
115 
116         while (true) {
117             System.out.println("Waiting for client messages to arrive");
118             Message msg;
119             try {
120                 msg = serviceInputPipe.waitForMessage();
121             } catch (Exception e) {
122                 serviceInputPipe.close();
123                 System.out.println("ServiceServer: Error listening for message");
124                 return;
125             }
126 
127             String ip = null;
128             try {
129                 Message.ElementIterator en = msg.getMessageElements();
130                 if (!en.hasNext()) {
131                     return;
132                 }
133                 MessageElement msgElement = msg.getMessageElement(null, "DataTag");
134                 if (msgElement.toString() != null) {
135                     ip = msgElement.toString();
136                 }
137                 if (ip != null) {
138                     System.out.println("ServiceServer: receive message: " + ip);
139                 }
140                 else {
141                     System.out.println("ServiceServer: error could not find the tag");
142                 }
143             }
144             catch (Exception e) {
145                 System.out.println("ServiceServer: error receiving message");
146             }
147         }
148     }
149 }
```

## C.2 The JXTA Client

```
1  package tutorial.service;
2 
3  import net.jxta.discovery.DiscoveryService;
4  import net.jxta.document.MimeMediaType;
5  import net.jxta.document.StructuredTextDocument;
6  import net.jxta.endpoint.Message;
7  import net.jxta.endpoint.StringMessageElement;
8  import net.jxta.peergroup.PeerGroup;
9  import net.jxta.pipe.OutputPipe;
10 import net.jxta.pipe.PipeService;
11 import net.jxta.protocol.ModuleSpecAdvertisement;
12 import net.jxta.protocol.PeerGroupAdvertisement;
13 import net.jxta.protocol.PipeAdvertisement;
14 import net.jxta.platform.NetworkManager;
15 
16 import java.io.IOException;
17 import java.io.StringWriter;
18 import java.io.File;
19 import java.util.Enumeration;
20 
21 public class ServiceClient {
22     static PeerGroup netPeerGroup = null;
23     static PeerGroupAdvertisement groupAdvertisement = null;
24     private DiscoveryService discovery;
25     private PipeService pipeService;
26     private NetworkManager manager;
27 
28     public static void main(String args[]) {
```

```
29          ServiceClient myapp = new ServiceClient();
30          System.out.println("Starting ServiceClient peer ....");
31          myapp.startJxta();
32          System.out.println("Good Bye ....");
33          System.exit(0);
34      }
35
36      private void startJxta() {
37          try {
38              manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC, "ServiceClient",
39                      new File(new File(".cache"), "ServiceClient").toURI());
40              manager.startNetwork();
41          }
42          catch (Exception e) {
43              e.printStackTrace();
44              System.exit(-1);
45          }
46          netPeerGroup = manager.getNetPeerGroup();
47          groupAdvertisement = netPeerGroup.getPeerGroupAdvertisement();
48          System.out.println("Getting DiscoveryService");
49          discovery = netPeerGroup.getDiscoveryService();
50          System.out.println("Getting PipeService");
51          pipeService = netPeerGroup.getPipeService();
52          startClient();
53      }
54
55      private void startClient() {
56          System.out.println("Start the ServiceClient");
57          System.out.println("searching for the JXTA-EX1 Service advertisement");
58          Enumeration en;
59          while (true) {
60              try {
61                  en = discovery.getLocalAdvertisements(DiscoveryService.ADV, "Name", "JXTASPEC:JXTA-EX1");
62                  if ((en != null) && en.hasMoreElements()) {
63                      break;
64                  }
65                  discovery.getRemoteAdvertisements(null, DiscoveryService.ADV, "Name", "JXTASPEC:JXTA-EX1", 1, null);
66                  try {
67                      Thread.sleep(2000);
68                  }
69                  catch (Exception e) {
70                  }
71              } catch (IOException e) {
72              }
73              System.out.print(".");
74          }
75
76          System.out.println("we found the service advertisement");
77          ModuleSpecAdvertisement mdsadv = (ModuleSpecAdvertisement) en.nextElement();
78
79          try {
80              StructuredTextDocument doc = (StructuredTextDocument) mdsadv.getDocument(MimeMediaType.TEXT_DEFAULTENCODING);
81              StringWriter out = new StringWriter();
82              doc.sendToWriter(out);
83              System.out.println(out.toString());
84              out.close();
85              PipeAdvertisement pipeadv = mdsadv.getPipeAdvertisement();
86              OutputPipe outputPipe = pipeService.createOutputPipe(pipeadv, 10000);
87              String data = "Hello my friend!";
88              Message msg = new Message();
89              StringMessageElement sme = new StringMessageElement("DataTag", data, null);
90              msg.addMessageElement(null, sme);
91              outputPipe.send(msg);
92              System.out.println("message \"" + data + "\" sent to the ServiceServer");
93          }
94          catch (Exception ex) {
95              ex.printStackTrace();
96              System.out.println("ServiceClient: Error sending message to the service");
97          }
98      }
99  }
```

# Appendix D

# Yapper

Follows is the Stage# implementation of Yapper as described in section 6.4. We omit the `Holder` actor as has no code beside the class declaration.

## D.1 Yapper `Client`

```
1  from Actors.keywords import *
2  from Demo.Yapper.person import Person
3  from Actors.Device.input import Keyboard
4
5  class Client(MobileActor):
6    def birth(self):
7      Keyboard().listen(self.shell)
8      print "commands:\n\t l (user) \t log in as the given user \n\t q \t\t quit"
9      self.attached = None
10
11   def shell(self, command):
12     opt = command[0:2]
13     param = command[2:len(command)]
14     if opt == "l ":
15       self.attached = Person(param)
16       here = loc(self.actor_id)
17       self.attached.attach(param, here)
18       self.help()
19     if opt == "q":
20        if self.attached is not None:
21          self.attached.migrate_away()
22        die()
23     if opt == "w" and self.attached is not None:
24       self.attached.print_wall()
25     if opt == "y " and self.attached is not None:
26       self.attached.yap(param)
27     if opt == "f " and self.attached is not None:
28         self.attached.follow(param)
29     if opt == "h":
30       self.help()
31
32   def help(self):
33     print "commands:\n\t w \t\t print wall \n\t\
34 y (message) \t yip a message \n\t f (user) \t Follow the given user \n\t\
35 h \t\t print this help message \n\t q \t\t quit"
36
37 def start():
38   Client()
39
```

```
40 if __name__ == '__main__':
41   initialise(start)
42 }
```

## D.2 Yapper Person

```
1  from Actors.keywords import *
2  from datetime import datetime
3
4  class Person(NamedSingletonActor):
5    def birth(self):
6      self.name = None
7      self.yips = list()
8      self.i_follow = set()
9      self.followers = set()
10     self.wall = list()
11
12   def attach(self, name, at):
13     if self.name is None:
14       self.name = name
15     if ip(self.actor_id) != ip_from_loc(at):
16       migrate(at)
17
18   def follow(self, friend_name):
19     friend = find_name(friend_name)
20     if friend.actor_id is None:
21       print("** '" + friend_name + "' is not on Yapper :-(")
22     else:
23       self.i_follow.add(friend)
24       friend.add_follower(self.name)
25
26   def add_follower(self, follower):
27     self.followers.add(follower)
28
29   def yap(self, yip):
30     now = datetime.now()
31     hour_min = str(now.hour) + ":" + str(now.minute)
32     yip = (hour_min, yip)
33     self.yips.append(yip)
34     self.updated('I', yip)
35     for person_name in self.followers:
36       person = find_name(person_name)
37       person.updated(self.name, yip)
38
39   def updated(self, name, yip):
40     self.wall.append("At " + yip[0] + " " + name + " yapped: " + yip[1])
41     if len(self.wall) > 10:
42       del self.wall[0:(len(self.wall) - 10)]
43
44   def print_wall(self):
45     for yip in self.wall:
46       print '+ ' + yip
47
48   def migrate_away(self):
49     holder = get_type("Holder")
50     theatre = loc(holder.actor_id)
51     migrate(theatre)
```

# Bibliography


[1] Jini Documentation. http://java.sun.com/products/jini/, 2005.

[2] Erlang/OTP Documentation R12B. http://www.erlang.org/doc/, 2007.

[3] JXTA Development Community. https://jxta.dev.java.net/, 2009.

[4] Gul A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. 1985.

[5] Gul A. Agha, Prasannaa Thati, and Reza Ziaei. Actors: a model for reasoning about open distributed systems. pages 155–176, 2001.

[6] D.P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 365–372, 2004.

[7] Apple Inc. Bonjour Technology Brief. http://images.apple.com/macosx/pdf/MacOSX_Bonjour_TB.pdf, 2005.

[8] Joe Armstrong. Concurrency Oriented Programming in Erlang. *Invited talk, FFG*, 2003.

[9] Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26. ACM, 2007.

[10] Joe Armstrong. A history of erlang. Talk, 2007.

[11] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[12] Joe Armstrong, B. Däcker, et al. Erlang white paper. http://erlang.org/white_paper.html.

[13] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 267–280. ACM, 1977.

[14] John Ayres. Implementing stage: the actor based language. MEng project final report, Imperial College, 2007.

[15] John Ayres. Private communication, 2009.

[16] John Ayres and Susan Eisenbach. Stage: Python with Actors. In *International Workshop on Multicore Software Engineering (IWMSE)*, May 2009.

[17] H.C. Baker Jr and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages table of contents*, pages 55–59, 1977.

[18] Bard Bloom and John Field. Thorn Cheeper Implementation. Private Communication, 2009.

[19] C.A. Bohn and G.B. Lamont. Load balancing for heterogeneous clusters of PCs. *Future Generation Computer Systems*, 18(3):389–400, 2002.

[20] A.R. Butt, R. Zhang, and Y.C. Hu. A self-organizing flock of condors. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Washington, DC, USA, 2003.

[21] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32. ACM Press, 1997.

[22] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *IN PROCEEDINGS OF THE 1ST NSDI*, pages 85–98, 2004.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10. USENIX Association, 2004.

[24] S.E. Deering. Host extensions for IP multicasting. RFC 1112 (Standard) http://www.ietf.org/rfc/rfc1112.txt, August 1989. Updated by RFC 2236.

[25] T. Desell, KE Maghraoui, and C. Varela. Load balancing of autonomous actors over dynamic networks. In *Proceedings of the Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track*, pages 1–10, 2004.

[26] A. Fisk. Gnutella Dynamic Query Protocol v0.1. http://www.limewire.com/developer/dynamic_query.html, 2003.

[27] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 118–128, 2003.

[28] M.J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-Transitive Connectivity and DHTs. pages 55–60, 2005.

[29] M.J. Freedman and D. Mazieres. Sloppy Hashing and Self-Organizing Clusters. *Lecture Notes in Computer Science*, pages 45–55, 2003.

[30] A. Fuggetta, GP Picco, G. Vigna, and D.E. e Inf. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

[31] L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[32] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394, 2003.

[33] J. Hautakorpi and G. Camarillo. Evaluation of DHTs from the viewpoint of interpersonal communications. In *Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*, pages 74–83, 2007.

[34] O. Hellström. Optimising TCP/IP connectivity. In *Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop*, pages 73–84, 2007.

[35] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[36] DG Kafura and KH Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages. *The Computer Journal*, 32(4):297–304, 1989.

[37] Kenneth M. Kahn. An actor-based computer animation language. In *UODIGS '76: Proceedings of the ACM/SIGGRAPH workshop on User-oriented design of interactive graphics systems*, pages 37–43. ACM, 1977.

[38] E. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 161–168, 2004.

[39] Eugene Letuchy. Facebook chat. http://www.facebook.com/note.php?note_id=14218138919, 2008.

[40] Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA '87: Proceedings on Object-oriented programming systems, languages and applications*, pages 17–34. ACM, 1987.

[41] S. Marlow, S.P. Jones, and S. Singh. Runtime Support for Multicore Haskell. 2009.

[42] R. Matei, A. Iamnitchi, and P. Foster. Mapping the gnutella network. *Internet Computing, IEEE*, 6(1):50–57, 2002.

[43] FJ Muniz and EJ Zaluska. Parallel load-balancing: An extension to the gradient model. *Parallel Computing*, 21(2):287–301, 1995.

[44] S. Oaks and L. Gong. *JXTA in a Nutshell*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002.

[45] P. Parnes, K. Synnes, and D. Schefstrom. Lightweight application level multicast tunnelling using mTunnel. *Computer Communications*, 21(15):1295–1301, 1998.

[46] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 SIGCOMM conference*, volume 31, pages 161–172, 2001.

[47] J. Ritter. Why Gnutella Can't Scale. No, Really. http://www.darkridge.com/~jpr5/doc/gnutella.html, 2001.

[48] R. Rodrigues and B. Liskov. High availability in DHTs: Erasure coding vs. replication. *Lecture Notes in Computer Science*, 3640:226–239, 2005.

[49] T. Rotaru and H.H. Nägeli. Dynamic load balancing by diffusion in heterogeneous systems. *Journal of Parallel and Distributed Computing*, 64(4):481–497, 2004.

[50] Arnon Rotem-Gal-Oz. The Eight Fallacies of Distributed Computing. http://www.rgoarchitects.com/Files/fallacies.pdf, 2004.

[51] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[52] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on Erlang*, pages 41–48, 2008.

[53] Jan Stender, Silvan Kaiser, and Sahin Albayrak. Mobility-based runtime load balancing in multi-agent systems. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'06)*, 2006.

[54] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 SIGCOMM conference*, pages 149–160, 2001.

[55] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. *IEEE/ACM Trans. Netw.*, 16(2):267–280, 2008.

[56] Y.M. Teo, V. March, and X. Wang. A DHT-based grid resource indexing and discovery scheme. 2005.

[57] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J.C. Hugly, E. Pouyoul, and B. Yeager. Project JXTA 2.0 super-peer virtual network. *Sun Microsystem White Paper*, 2003.

[58] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, 2001.

[59] Jim Waldo. The Jini architecture for network-centric computing. *Commun. ACM*, 42(7):76–82, 1999.

[60] MH Willebeek-LeMair, AP Reeves, I.B.M.T.J.W.R. Center, and Y. Heights. Strategies for dynamic load balancing on highly parallel computers. *Parallel and Distributed Systems, IEEE Transactions on*, 4(9):979–993, 1993.

[61] B.J. Wilson. *JXTA*. New Riders Boston, 2002.

[62] Dave R. Wisely, Louise Burness, and Philip Eardley. *IP for 3g: Networking Technologies for Mobile Communications*. John Wiley & Sons, Inc., 2002.

[63] Tobias Wrigstad, Gregor Richards Johan Östlund, Jan Vitek, Bard Bloom, John Field, Nathaniel Nystrom, and Rok Strnisa. Thorn–Robust, Concurrent, Extensible Scripting on the JVM. Draft Copy, 2009.

[64] Z. Xu, C. Tang, and Z. Zhang. Building topology-aware overlays using global soft-state. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 500–508, 2003.

[65] Charles Ying. What you need to know about Amazon SimpleDB. http://www.satine.org/archives/2007/12/13/amazon-simpledb/, 2007.

[66] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, 2001.

[67] B. Zuhdy, P. Fritzson, and K. Engstroem. Implementation of the real-time functional language Erlang on a massively parallel platform, with applications to telecommunications services. *Lecture Notes in Computer Science*, pages 886–886, 1995.