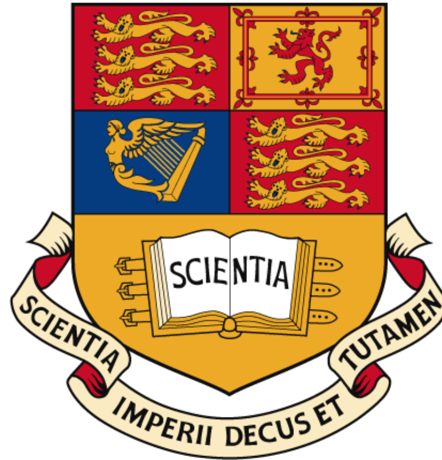
Imperial College of Science, Technology and Medicine
Department of Computing



A String of Ponies

Transparent Distributed Programming with Actors

Sebastian Blessing

Supervised by Prof. Sophia Drossopoulou and Sylvan Clebsch

Submitted in partial fulfilment of the requirements for the MSc Degree
in Computing Science (Software Engineering) of Imperial College London,
September 2013

Abstract

We develop an extension to a concurrent, actor-based language runtime called *Pony* with the ability of *transparent* distributed programming, whereby programmers do not need to be aware of the underlying distributed system. Thus, any *Pony* application can be executed in a concurrent setting as well as in a distributed setting without any changes to the code being necessary.

A distributed cluster of *Ponies*, managed based on a tree network topology, is easy to maintain and can be extended with slave nodes at runtime without any reconfiguration being necessary. We propose a joining algorithm which guarantees that the underlying tree topology is *almost* balanced at any point in time.

Distribution is reflected through *actor mobility*. We develop a hierarchical work stealing algorithm with constant space complexity, specifically tailored for tree network topologies. Work stealing is provably optimal for task and data parallelism. Furthermore, the implementation of the proposed distributed work stealing algorithm is itself based on the actor programming model, which allowed us to extend *Pony* without sacrificing the performance of single-node configurations.

Causal message delivery is a valuable property for message-passing systems, contributing to efficiency and improved reasonability. *Pony* guarantees causality for both the concurrent and distributed setting without any additional software overhead.

A property unique to *Pony* is fully concurrent garbage collection of actors. We present an extended algorithm for concurrent garbage collection (including cycle detection) in distributed actor-based applications. Concurrent *Pony* programs are terminated based on quiescence. In order to retain that property in a distributed context, we propose a protocol for detecting quiescence in distributed actor systems. Both schemes strongly depend on causal message delivery.

We evaluate our implementation based on a micro benchmark that allows us to simulate different application characteristics. *Pony* achieves considerable speed-up for computation-bound scenarios.

Acknowledgements

I would like to acknowledge the support and input from a number of people who helped me to bring this thesis into being.

Firstly, my two supervisors, Prof. Sophia Drossopoulou and Sylvan Clebsch, deserve my thanks for guiding me through this project, for investing more time than I could have ever expected and for offering me the opportunity to work on a programming language runtime. I would also like to thank them for changing my perspective on the importance of type systems for reasonability and efficiency, as well as for influencing my view on what future programming languages could be like.

I would also like to thank Prof. Alexander Wolf for having increased my interest in distributed systems and algorithms as well as for preparing me with the necessary knowledge for a project like this, which helped me to identify and overcome the challenges that came up during the development of *Distributed Pony*.

I owe a huge debt to my parents, Dieter and Monika, for having supported me in pursuing my dreams. Thanks for introducing me to the world of computers at a very young age and for allowing me the explore my interests. Without them, I would have never reached this point. I must also thank my siblings, Alexander and Ann-Kathrin, for sharing their experiences, which made so many things easier for me.

I thank Reuben Rowe for having mentored me throughout this Masters program. I would like to extend my appreciation to the SLURP group of Imperial College London for the interesting discussions on various research topics of computer science.

I thank Florian Oswald for his friendship and for joining me in founding a company. I am looking forward to developing innovative software systems, solving interesting problems and seeing our company evolve.

Finally, I would like to thank Prof. Dr. Werner Zorn for having provided me with advice throughout my entire university career as well as Prof. Dr. h.c. Hasso Plattner for supervising my undergraduate studies at HPI Potsdam and for supporting me in evolving as a software engineer.

Contents

Abstract i

Acknowledgements iii

1 Introduction 1

2 Background 3

2.1 Programming Distributed Systems 4

2.2 The Actor Programming Model 5

2.3 Causality in Distributed Systems 6

2.4 Failure Detection in Asynchronous Networks 7

2.5 Scheduling of Tasks 9

2.5.1 Preemptive and Cooperative Scheduling in Operating Systems 9

2.5.2 Priority-based Scheduling 11

2.5.3 Operating System Scheduling vs. Actor Runtime Scheduling 11

2.5.4 Work Stealing 12

2.6 Pony 14

2.6.1 The Pony Actor 15

2.6.2 Message Queues 18

2.6.3 Work Stealing Queue 20

2.6.4 The Scheduler 22

2.6.5 Pool Allocator 23

2.6.6 Garbage Collection 23

2.6.7 Termination 24

2.7 Available Actor Languages and Runtimes 25

2.8 Conclusions 26

3 Distributed Pony 27

3.1 Overview of Runtime Components 28

3.2 Asynchronous I/O Multiplexing 29

3.2.1 Epoll, KQueue and Kevent 30

3.2.2 Select and Poll 30

3.2.3 Scheduling I/O Events 33

3.2.4 Framed Network Protocol 34

3.3 The Distribution Actor . . . 36
3.3.1 Non-blocking Sockets . . . 37
3.3.2 Connecting Slave Nodes and Routing . . . 39
3.4 Serialization and Deserialization . . . 41
3.4.1 Stream Buffer and I/O Vector . . . 42
3.4.2 Pony Trace . . . 43
3.5 Distributed Work Stealing . . . 45
3.5.1 Migration and Actor Proxies . . . 46
3.5.2 Causality-aware Migration Protocol . . . 48
3.5.3 Handling Actor References and Actor Identity Comparison . . . 50
3.5.4 Hierarchical Work Stealing . . . 52
3.5.5 Collapsing Actor Proxies . . . 54
3.6 Distributed Object Identity Comparison . . . 55
3.7 Distributed Garbage Collection . . . 57
3.7.1 Deferred Reference Counting and Passive Object Collection . . . 57
3.7.2 Active Object or Actor Collection . . . 58
3.7.3 Centralized Cycle Detector . . . 60
3.7.4 Hierarchical Cycle Detection . . . 61
3.8 Termination . . . 63

4 Causality in Tree Network Topologies **66**
4.1 Informal View . . . 66
4.2 Formal Argument . . . 67

5 Evaluation **68**
5.1 Test Application . . . 69
5.2 Message-bound vs. Computation-bound . . . 70
5.3 Programming Example - Mandelbrot Fractals . . . 73

6 Conclusion **77**
6.1 Contributions . . . 77
6.2 Future Work . . . 79

A Non-blocking reads **81**
A.1 Pseudocode - socket read handler: collect() . . . 81

B Micro Benchmark **82**

Bibliography **85**

Chapter 1

Introduction

In the last several years, the development of computer systems has come to an inflection point – the *"free lunch is over"* [117]. Increasing demands on complex software applications integrating multiple applications and various data sources for potentially massive amounts of users require to utilize the underlying hardware resources as efficiently as possible. However, the amount of computing power that can be put into a single machine is physically limited. As a consequence, applications need to be able to utilize computation resources from multiple machines that are (possibly) distributed over a large network.

Programming languages are the interface between software developers and the underlying hardware resources. Not only is it important to develop languages that provide levels of abstraction for easier development and improved maintainability but also to provide runtime systems that utilize available hardware resources efficiently. This should be a property that holds also for a distributed set of computers within a cluster and not only for locally available resources. High levels of abstraction and high performance must not necessarily be contradicting requirements for a language runtime but it is difficult to achieve both goals. It is important in the design process of such a language to balance design decisions along both goals which may require prioritization of features. Thinking for the long term is extremely vital and therefore design decisions need to be evaluated in terms of the impact they have on upcoming features in future versions. Furthermore, once decisions have been made, the implementation is still a challenging task to accomplish.

Pony, developed at Imperial College London, is an actor-model [63, 2], object-oriented, strongly-typed programming language for high performance computing. Concurrency is reflected through sending *asynchronous* messages between actors and dynamically *creating* new actors [2]. Incoming messages are buffered in an actors *mailbox*. One important property of *Pony* is causal message delivery. Each message is an *effect*, and previously sent or received messages are a *cause* of that effect [30]. Causal message delivery guarantees that the cause of a message is always enqueued in an actors mailbox before its effect [30]. Besides being helpful for reasoning about programs, this property is crucial for a feature unique to *Pony*: fully concurrent garbage collection of actors [30].

The motivation for programming based on message-passing is to encourage the decomposition of applications into self-contained autonomous components. Components are independent from each other and there is no need to care about data races within applications, which should ultimately aid in ease of development and maintainability. At the same time, because low-level synchronization for parallel applications is not required to be done by the programmer, efficiency can be significantly improved depending on the underlying runtime implementation (and type system). The Actor model is inherently well suited for distributed computing [2].

The goal of this project is to enable *Pony* for transparent distributed programming. We refer to it as transparent programming, because the degree of distribution is not exposed to the programmer. Any application written in *Pony* should scale in a distributed network of runtime process without *any* changes to the code being necessary. This task is challenging, because we want to achieve that *any* conceptual property given by *Pony* in the concurrent setting also holds in a distributed context.

Our approach for *transparent* distributed programming builds upon *actor mobility*. We propose an implementation of a distributed scheduler that allows to efficiently distribute actors in a network of computers. It is desirable to utilize a computer cluster in a balanced manner, rather than having unused resources running in idle state. The scheduling mechanism used for the *Pony* runtime is known as *work stealing* [15, 37, 106, 101]. Local processor cores and (in the distributed setting) cores from remote machines that run in idle state ("thieves") *steal* work from busy nodes ("victims") [15].

The challenge is to extend *Pony* for distributed computing without sacrificing the concurrent performance or trading off competing factors against each other between concurrency and distribution. Preferably, both parts should be entirely independent from each other. This is difficult because it is not sufficient to just implement a scheduler, we also need to guarantee that the language semantics of *Pony* also hold in the distributed setting, such as garbage collection of actors and passive objects, distributed termination, causal message delivery as well as object and actor identity comparison.

This thesis provides the following contributions:

- An algorithm with constant space complexity for hierarchical work stealing in tree networks.
- A joining algorithm to add new slave nodes to a cluster of *Ponies* at runtime.
- Deferred reference counting in distributed systems.
- Distributed hierarchical garbage collection of actors.
- Causal message delivery in distributed systems with no software overhead.
- Termination of actor applications based on distributed quiescence.

Outline of the Thesis

This thesis is split up into three parts. Chapter 2 motivates the need for a new programming language for concurrent and distributed computing, gives an overview of the actor model and discusses the challenges of distributed programming. After having provided an overview on scheduling of tasks, we discuss implementation details of the concurrent *Pony* runtime.

Chapter 3 describes the implementation of *Distributed Pony*. This includes a distributed scheduler (section 3.5), object identity comparison (section 3.6), garbage collection in section 3.7 and a scheme for distributed termination of actor-based applications (section 3.8). *Distributed Pony* guarantees causal message delivery with no software overhead. An informal view on the problem of causality in distributed systems as well as a formal argument for *Pony's* causality property is provided in chapter 4.

We explicitly decided not to implement our own benchmark suite for *Pony*. Instead, chapter 5 evaluates the performance of *Pony* for computation-bound and message-bound scenarios. We conclude and give an outlook for future work in chapter 6.

The algorithms and protocols discussed in this thesis have been developed in collaboration with Sylvan Clebsch and Prof. Sophia Drossopoulou.

Chapter 2

Background

Distributed Computing focuses on the development of software systems where different components are located on physically distinguishable machines that are connected to a network of computers [34]. Communication is implemented by sending messages between components using a network protocol such as TCP [22, 69, 71]. Components in this context does not only mean independent applications and services are integrated to a new system, but also parts of the same application which run on different machines in the network, where each part is critical to the overall application or is used as a replication factor. Thus, the motivation for developing distributed applications is *scalability* and *fault-tolerance*. In this work, we focus on the programming of distributed applications rather than on solving specific problems in an asynchronous network of processes (e.g. consensus or asynchronous replication).

According to Coulouris et al, the main characteristics of such a system are concurrency of components, the lack of a global notion of time, as well as independent and partial failure [34]. Technically, the main difference between a parallel system and a distributed system is that the former consists of a set of processors that share the same virtual memory to exchange information, whereas independent components running on different machines need to exchange information using message passing, as there is no shared memory.

Evidently, programming languages are required to allow for the development of *distributed programs*. How distribution can be expressed within programs and to which extend the programmer is exposed to the physical topology of a network of computers differs largely between languages and also depends on the programming model used. Section 2.1 motivates the need for programming language support for distributed systems. The Actor model is the basis of *Pony* and is therefore covered in more detail in section 2.2.

To which extent the lack of global time in distributed systems is a problem and why causality might be desirable – and in fact is required in the context of *Pony* – is discussed in section 2.3. *Asynchronicity* through sending messages between components comes with the problem of partial failure. As *Distributed Pony*, developed for this project, transparently migrates actors from one machine to another, this problem also applies for the implementation of a programming language runtime. Although failure detection is not the main topic of this work, it shall not remain unmentioned in this thesis. Thus, section 2.4 gives an informal view on the problem.

The remainder of this chapter provides a short description of techniques for assigning CPU time to entities of execution (i.e. scheduling) as well as a detailed overview of the single-node implementation of *Pony*, which is the basis of this work. Note that this project focuses on the runtime implementation and therefore a description of the *Pony* syntax and the operational semantics is not provided.

2.1 Programming Distributed Systems

Network technologies, the upcome of the Internet and scalability requirements of large software systems brought about the need for distributed programming. Although distributed systems have been a research topic for many years [78, 62, 105, 57], the "*fallacies*" [36] of these systems and their programming are often underestimated [74].

Many approaches to support distributed programming have been proposed. Kendall et al observed [74], that especially for object oriented programming, many of these approaches aimed in integrating program distribution seamlessly, such that local and remote objects are treated (almost) the same. A widely-used implementation of this idea is Java RMI (*Remote Method Invocation*) [68], which is a high level and object-based implementation of remote procedure calls (RPCs) [123, 10]. However, the fact that methods of remote objects are called in the same way as methods of local objects is problematic. It creates the illusion that invoking a method on a remote object compared to a local object has no effect on a program's correctness. Similar approaches prior to Java RMI, such as CORBA [54], suffer the same weakness.

The problem is *partial failure*. For example, a remote invocation can fail before the call was executed or it has been executed at the remote site but the response got lost. Whereas the former can be handled with timeout mechanisms provided by the underlying network protocol, the latter is more complicated, because the client side cannot detect whether the method was invoked or not. The implementation of the remote interface would require to cater for that, which somehow contradicts the idea of treating remote and local objects the same way. Furthermore, reasoning about such programs is difficult [74].

Lamport proposes that the problem of partial failure – among others – can be solved based on *Paxos* [77], a family of algorithms developed for various purposes in the context of distributed systems. The idea is to implement a distributed state machine that caters for partial failure by construction. Although it is a valid solution, this approach does not help in simplifying the programming of such systems, or increasing the programmers productivity.

Evidently, language support (not just library support) for the development of reliable distributed applications is indispensable. Prominent examples of languages that were specifically designed for concurrent *and* distributed programming tend to be functional in nature (e.g. *Erlang* [38] or *Mozart* [31]). However, functional programming is too restrictive and feels inconvenient. Erlang implements the Actor programming model (but calls them processes). However, distributed applications are built in way that the degree of distribution and participating nodes are actually visible to the programmer. The fact that the distribution is directly embedded into the code makes it more difficult to program applications that scale dynamically on any kind of computer cluster. *Scala* [40], another object-based and functional language (at least syntactically), also implements the Actor paradigm, but is not suitable for distributed computing without extensive use of external libraries such as Apache Thrift [3].

Motivating the need for a new programming language is difficult. However, in the case of *Pony* and compared to available languages based upon the Actor paradigm, we believe that developing a new language is an exciting endeavor to undertake (for both concurrent and distributed programming). Furthermore, our approach to first develop the runtime and then provide a language that fits it, is radically different. Although future work, the design of a type system specifically tailored for the developed runtime can aid in efficiency and programmer productivity to a large extent. At the same time, next to the programming model chosen, a type system contributes to the way we reason about programs. We believe that important contributions can be made to both the way we program and how we reason about concurrent and distributed applications.

2.2 The Actor Programming Model

There are many different ways in designing a programming language. One important step is to decide whether a language should be intrinsically connected to a programming model or should be as generic and unbiased as possible to provide the maximum degree of flexibility. The choice of the programming paradigm affects the style of programming and possible pitfalls in program development (e.g. inefficient programming).

In the context of today's requirement of high performance, we need to choose a paradigm that supports parallelism. How programs are mapped to multi-core hardware resources is not only important for programming convenience but also vital for reasoning about programs and debugging. It is questionable if using *threads* for concurrency directly within a language is an appropriate solution to these problems. The need for synchronization mechanisms to avoid races makes developing applications fundamentally error prone and unnecessarily complex for the programmer and distract from the actual problem to solve. Additionally, the fact that the provided degree of fine-grained synchronization leads to the opportunity to reach a maximum degree of efficiency and concurrency within a system is disputable. Parallelizing compilers or high level synchronization primitives (e.g. monitors) are not satisfying due to the fact that they inherently do not reach a high degree of concurrency [64, 58, 59, 20, 19, 13]. Type systems that guarantee race-free execution could be an answer to many of the problems described, but require the programmer to understand and write (possibly) complex type annotations [18, 29, 35, 43].

One of the goals set for the *Pony* programming language and this project is to show that using a model that provides high levels of abstraction can be implemented in an efficient manner whilst providing high performance and safety at the same time without sacrificing programming convenience. Similar to the theoretical studies discussed in Tony Hoare's book "*Communicating Sequential Processes*" [65], the programming model used for *Pony* is based on an abstract interface (called an Actor) for switching the state of objects by sending asynchronous messages (usually point-to-point). This is called the Actor programming model, proposed by Hewitt et al in 1973 [63, 2].

Each actor maintains its own heap to which no other actor holds references to. This means that the state itself is contained and protected within the actor and can only be manipulated through messages. For this purpose, each actor has a unique identifier (address) and maintains it's own message queue to buffer incoming messages from other actors. Adding a message to this queue must be an atomic operation. This approach entirely removes the need for shared memory synchronization. The only point of synchronisation is an actor's message queue, which can be implemented efficiently with lock-free atomic memory operations such as *k-word Compare-and-Swap* [55] or *Test-and-Set* [52]. Furthermore, from a language perspective, there is no difference between an actor being located on the same machine or on a remote host. This makes the Actor programming model inherently suitable for distributed computing. An actor consists of three attributes/actions [63]:

- Send a finite number of asynchronous messages to other actors
- Receive messages from other actors and react based on the type of a message
- Create a finite number of new actors

As we intend to support the transparent sending of messages between spatially separated actors, guaranteed delivery needs to be considered by the *Distributed Pony* runtime and the underlying network stack. From a programming perspective, implementing parallel, concurrent and distributed applications through sending messages between various components of the system (which are implemented as actors) is a natural way of thinking and results in a modular software design with loose coupling. Furthermore, as race conditions are not a matter in this context, and if causal message delivery can be guaranteed (which is not a part of the actor model itself, see section 2.3), debugging and reasoning about programs based on this model can be significantly improved [110].

Note that because communication is implemented through messages; values, variables or objects could possibly be located in the heap of several actors at the same time which might result in

memory overhead. This is due to the nature of the message passing paradigm, where objects are usually handled as value types instead of referenced objects that could be owned by another process. Within this work, we will mainly focus on the implementation of actors in the runtime rather than putting it into a context of theoretical foundations of message-passing, which are discussed in more detail in [63, 65, 102, 103].

In *Pony*, actors are the structure that the runtime scheduler is operating on. It will be shown in this work that employing the actor programming model does not only have positive effects from a programmers perspective who uses the *Pony* language, but also on efficient scheduling of network events as well as garbage collection.

2.3 Causality in Distributed Systems

The non-deterministic nature of distributed systems and the lack of global time and state make development and debugging of distributed applications a challenging task. Reasoning about distributed programs and algorithms strongly depends on the ability of observing the causal order of events that *as a whole* carry out a computation task [110, 108]. Besides failure detection and reproducibility in the context of such systems [24, 97], causality is one of the most important issues for developing reliable and efficient asynchronous distributed systems (e.g. debugging [110], global state detection [25], efficient communication protocols [8, 9] as well as *Pony's* concurrent garbage collection mechanism [30]).

Distributed systems are commonly modeled as a set of $n \in \mathbb{N}$ (usually sequential) processes $p_1, ...p_n$ which are spatially separated and communicate only through messages [110, 76]. Generally, Lamport classifies a system as distributed, if the message transmission delay from one process to another is not negligible compared to the time between the occurrence of two events in a single process [76]. Lamport introduced the notion of logical time [76] to obtain a time diagram of events in a distributed system. The idea is that maintaining an order of events is possible without the availability of perfectly synchronized clocks between all participants. Thus, time diagrams of events that can be obtained with logical clocks are an equivalent representation of the events from the perspective of distributed computation (i.e. a trace of the progress made within the system). A precise time ordering is not necessary as it is biased by unnecessary effects like network delay and relative speeds of independent processors. Hence, the *causal* order of events is sufficient for reasoning about distributed programs.

For this purpose, Lamport introduced the causality relation "*happens before*" [76] to impose a partial order on a set of events. Rather than implementing causality through physical clocks, logical time is sufficient to check whether a given program meets the specification in terms of events that can be observed during runtime [76]. Furthermore, using physical clocks would require some synchronization mechanism, which is unnecessary overhead [76]. An event e_1 *happens-before* another event e_2 (denoted as $e_1 \rightarrow e_2$), iff [76]:

1. e_1 and e_2 belong to the same (sequential) process and e_1 *precedes* e_2 or,
2. e_1 is the sending of a message in process p_1 and e_2 is the receipt of that message in process p_2.
3. $e_1 \rightarrow e_2 \land e_2 \rightarrow e_3 \Rightarrow e_1 \rightarrow e_3$ (transitivity)
4. $e_1 \not\rightarrow e_1$ (irreflexive)

Events that cannot be correlated to each other using the above rules are said to be concurrent (i.e. neither causally effects the other) [76]. Logical clocks can be implemented through having each process maintaining a local clock value (e.g. a simple integer). Thus, if $e_1 \rightarrow e_2$, then the corresponding clock values C should obey $C_{e_1} < C_{e_2}$. This is guaranteed if each process "*ticks over*" [76] the local clock value at each occurrence of a local event. If an event is caused by the receipt of a message from another process p', then the clock value must be set to $max(C_{local}, C_{p'} + 1)$. This requires every message to contain the local clock value of the sending process. Note that

$C_{e_1} < C_{e_2}$ does *not* imply $e_1 \rightarrow e_2$, because local clock values are independent between processes that never communicated. This is a problem, because it is not possible to determine events that are causally independent by simply comparing their clock values. However, this problem can be solved by using vector clocks [96, 41], where each process maintains a vector of clock values with as many elements as processes in the system. This causes that clocks of processes that never communicate get eventually "*synchronized*". The rules mentioned above must simply be applied per element. The negative impact of vector clocks on performance is self-evident, especially in systems with many processes/nodes.

The garbage collection mechanism developed for *Pony* strongly depends on causal delivery of messages. Furthermore, the examples mentioned above show that such a property is highly desirable for a programming language that supports distributed computing. It seems complex and expensive to provide a runtime implementation that enforces causal delivery. In fact, maintaining vector clocks includes substantial computational overhead and even with properly maintained clocks, reconstructing the causal relation of events is expensive [110, 27, 111].

It has been previously mentioned that designing a programming language with certain properties (e.g. causality) and achieving high performance should not be contradicting requirements but can be difficult to achieve at the same time. Guaranteeing message causality is key to providing a language where distributed computation is transparent to the programmer, such that programs running on a single machine can be executed on arbitrarily many nodes without any changes to the code being necessary or that the programmer needs to be aware of the underlying hardware and network topology. However, this degree of transparency is only sensible if *Distributed Pony* can cope with failures appropriately, because distributing computation comes with the cost of partial failure.

2.4 Failure Detection in Asynchronous Networks

A consequence of implementing the Actor programming model is that communication between program components is reflected through *asynchronous* messages. Scheduling actors on remote nodes also implies that nodes can fail independently from each other. The result given by Fisher, Lynch and Patterson (FLP-result [42]) states that it is impossible to solve the distributed consensus problem in an asynchronous network if there is a single faulty process and failures cannot be detected. This result can be generalized to almost any problem that one might solve in the distributed context [17]. Hence, a discussion about the implications this result has on *Distributed Pony* is required and to which extent partial *failure* needs to be addressed by the runtime implementation (beyond guaranteed delivery of messages).

Chandra and Toueg [24, 23] show that a way out of this is to implement *failure detectors* (which themselves can be unreliable) to implement reliable distributed systems. A failure detector is essentially a software module that can be used by the program to detect failures. Since failures are difficult to detect reliably (because a process could just be extremely slow), the terminology in use is *suspecting* a process rather than detecting that it had failed. Applications are then built based on the fact that there is no perfect knowledge about the current state of a process, but suspicions are enough to keep operating. Conceptually, a failure detector allows each process to maintain a set of suspected processes. Note that because a detector is local to each process, distinct detectors might have different views and suspect different processes.

A function $F : \mathbb{T} \rightarrow 2^{\Pi}$ that assigns to any instance of time $\mathbb{T}$ a set of suspected processes is called a *failure pattern* ($\Pi = \{p_1, ..., p_n\}, n \in \mathbb{N}$) [24]. The failure detector as a module outputs the set of processes that it currently suspects to have crashed. Note that $\mathbb{T}$ is not physical time or visible to any of the processes, it is just a means of modeling the problem domain. In the context of an asynchronous system, where network delay or the relative speeds of independent processors are unknown and a Gaussian delay is possible, suspicions may differ from time to time (i.e. a process might be suspected at time t_1 but not suspected at any time instance after t_1). Thus, the set of all processes that have been suspected over time is formalized as a *failure history* $H : \Pi \times \mathbb{T} \rightarrow 2^{\Pi}$ [24]. A process p *suspects* q at time t, if $q \in H(p, t)$. The asynchronous nature of

the system in observation and the fact that each process has a local failure detector module causes that suspicions can vary in quality. This means that if $p \neq q, H(q,t) \neq H(p,t)$ is possible. Hence, for reasoning purposes, the set of failure detectors is modeled as a non-deterministic detector $\mathbb{D}$ that maps failure patterns to a collection of failure detector histories for that pattern [24]. A concrete technical specification for a generic detector would be too complex as it needs to consider many parameters (such as the network protocol in use, network delays etc.), which might not be known in advance. However, the properties *completeness* and *accuracy* are sufficient to reason about applications that use a specific failure detector. This also means that the knowledge of the concrete detector in use is essentially "embedded" in the algorithm/program that depends on it.

Chandra and Toueg classify the following properties of detectors [24]:

- Let $C(F) = \bigcup_{t \in \mathbb{T}} F(t)$ be the set of crashed processes and $R(F) = \Pi - C(F)$ the set of correct processes. Assume that crashed processes never recover.
- *Completeness* states that a detector will suspect crashed processes. The detector can suspect correct processes, but does not miss any crashed ones. A detector is said to be *strongly* complete if every crashed process is permanently suspected by every correct process:

$$\forall F, \forall H \in \mathbb{D}(F), \exists t \in \mathbb{T}, \forall p \in C(F), \forall q \in R(F), \forall t' \geq t : p \in H(q,t')$$

- *Accuracy* ensures that correct processes are not suspected. There is no guarantee that the detector suspects crashed processes, only that it does not suspect correct ones. Similar to above, a detector is *strongly* accurate if a process is not suspected if it has never crashed:

$$\forall F, \forall H \in \mathbb{D}(F), \forall t \in \mathbb{T}, \forall p, q \in \Pi - F(t) : p \notin H(q,t)$$

Both properties also exist in a *weak* form, where a detector is *weakly* complete if all faulty processes are suspected by *some* correct processes [24]. Similarly, a detector is *weakly* accurate if there is *some* correct process that is never suspected [24]. Furthermore, both strong and weak accuracy guarantee that there is at least one correct process that is never suspected. Technically, this is difficult to achieve in an asynchronous system [24]. For this reason, both types of accuracy are usually used in their temporal form, such that the corresponding property holds after some point in time (but after that point in time it holds forever). This is also referred to as eventual strong/weak accuracy [24].

It is self-evident that these properties alone are not sufficient. A complete failure detector can simply suspect all processes ($\forall t \in \mathbb{T}.H(q,t) = \Pi$). Furthermore, a detector is still accurate if it does not suspect any processes ($\forall t \in \mathbb{T}.H(q,t) = \emptyset$). Hence, a detector is only useful if it combines completeness and accuracy in some form or another. From the six properties mentioned above, Chandra and Toueg derive eight classes of failure detectors by simply picking one of the completeness properties and one of the four discussed types of accuracy, as shown in table 2.1. Each detector class can then be used as an assumption based on which an algorithm or application is developed.

The combination of weak completeness and strong accuracy as well as the combination of the corresponding temporal forms is left empty, because technically there would be no reason not to use their perfect counterparts, as these types of failure detectors guarantee a subset of processes whose local detector is perfect by construction.

From a programming point of view, a failure detector is then probed when a process is expecting to receive messages from one or more processes (the number is depending on the algorithm/program itself). One implementation could be to let a *collect* statement return false if the process from which an incoming message becomes suspected, otherwise the message can be delivered to the process. Failure detectors are software modules and local to each process (or in the case of *Pony* to each actor or node). So, they can be implemented in *Distributed Pony* as part of the application in order to solve problems in the distributed context. For this reason, the problem of failures from a runtime point of view reduces to the problem of guaranteed message delivery. As a consequence, failure *detection* is considered as future work and will be re-considered in section 6.2. Work on concrete distributed algorithms using failure detectors has been published in [49, 114].

Table 2.1: Eight classes of failure detectors [24]

Completeness	Accuracy			
	strong	weak	eventually strong	eventually weak
strong	Perfect	Strong	Eventually Perfect	Eventually Strong
weak		Weak		Eventually Weak

2.5 Scheduling of Tasks

Scheduling is a well studied topic in Computer Science, especially in the context of process scheduling for operating systems [112, 113, 56] and distributed systems [11, 21, 53, 121, 107, 109]. The field of resource allocation problems is a vast subject and therefore we will mainly concentrate on the basics and those mechanisms that are of importance for the runtime implementation of *Pony*. In the following, a short overview of various scheduling mechanisms is given. It is worth studying the concepts of operating system scheduling, because for a programming language runtime we pursue similar goals.

In operating systems, the objective is to have some process running at every point in time, such that the CPU spends as less time as possible in idle state. Observations [112] have shown, that processes usually alternate between CPU-bursts and I/O-bursts. From the perspective of a processor, I/O events can require many cycles for completion. During this time, the invoking process has to wait. This obviously wastes CPU cycles, because the processor remains unoccupied during that time. The purpose of a scheduler is to ensure that this time is used productively. Many criteria for characterising different scheduling mechanisms have been proposed [112]:

- Maximize **CPU utilization**.
- Maximize **throughput**, i.e the number of processes that complete their task per time unit.
- Minimize **turnaround time or latency**, which is the time difference between the submission of a process and its completion.
- Minimize the time that a process has to **wait** in a queue for ready processes.
- Minimize **response time**, which is the time from submission of a process until the first result is displayed to the user.

The way how access to system resources is managed and which of the above criteria are chosen as optimization goal largely depends on the characteristics of the environment a system is operating in as well as application requirements. In the context of this work, we will only discuss *dynamic* scheduling, where the schedule is determined at runtime as opposed to *static* scheduling, where it is computed before execution (e.g. for predictable and time critical real-time systems).

2.5.1 Preemptive and Cooperative Scheduling in Operating Systems

Processes are commonly described as a state-machine with four states, as shown in Figure 2.1[1]. Dashed lines indicate that a process switches its state due to some system event that is not in the control of the process. Continuous lines illustrate state transitions that are initiated by the process itself.

Generally, scheduling needs to take place whenever a process transits from one state to another. This happens when any of the following events occur [112]:

1. A process **terminates**.
2. A process actively **waits** for **I/O events**.
3. Requested **I/O** operations are **completed**.

[1] The diagrams for this thesis have been created using *yed* [124]

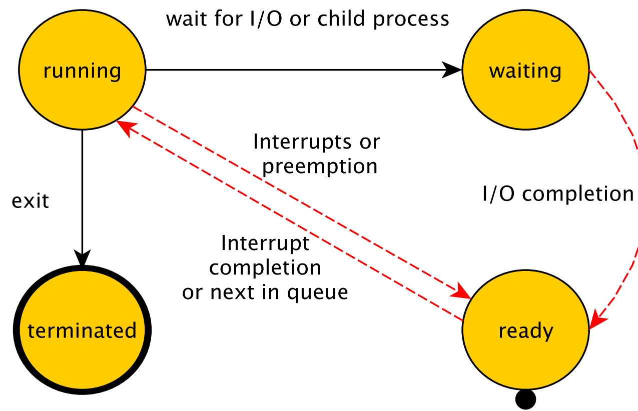


Figure 2.1: State Transitions of Processes

4. A process is **preempted** because it used up its *quantum*, or **hardware interrupts** need to be handled.

5. **Interrupts** are handled or the process is chosen next.

In the context of operating systems, a scheduling mechanism is called *cooperative*, if it only takes place when a process explicitly signals that there is no work to-do (in cases (1) and (2)) [112]. A mechanism that triggers the scheduler additionally in response to the cases (3) and (4) is called *preemptive* [112]. Note that case (3) is appointed to be preemptive because it leaves a choice to the scheduler to preempt the current process if priorities are involved (see section 2.5.2), whereas cases (1) and (2) simply cause the *next* process to be scheduled (the meaning of the term *next* is further discussed in the remainder of this section).

Although there only seems to be a small difference between the two schemes, their effect on the design of an operating system is quite dramatic. Cooperative scheduling can be the only choice possible if the underlying hardware architecture does not support switching the context between a running process and a ready process. *Preemption* comes with the cost of shared memory synchronization [112]. Moreover, the operating system kernel has to cope with system calls that return after a process has been preempted. The implications of scheduling mechanisms on the design of an operating system kernel are further discussed in [112].

A simple example for a cooperative scheduling scheme is *first-come, first-served* (FCFS) [112]. All processes are appended to the tail of the same FIFO queue. Upon process termination or when a process enters its I/O phase, the scheduler simply allocates the process at the head of the queue to the CPU. In a system where processes are all of the same kind (in terms of their CPU- and I/O-burst pattern) this scheme might suffice. However, the cooperative nature of this scheme might cause the average waiting time to greatly differ if CPU intensive processes are scheduled prior to relatively short ones in an alternating fashion. This effect can be cushioned using *shortest-job-first* (SJF) scheduling [112]. As the name suggests, the idea is to schedule the process with the shortest CPU-burst time first; ties are broken using FCFS scheduling. If minimizing the average waiting time of a process is of highest priority, then this scheduling mechanism is provably optimal [112]. Intuitively, moving a process which is less CPU-intensive in front of comparatively long ones has less of an effect on the waiting time of long processes as compared to the waiting time when moving long processes in front of short ones. However, the time interval of CPU-bursts for each process needs to be known or estimated a priori. This is unrealistic in a generic execution environment. We will re-elaborate on this statement in the context of a language runtime scheduler in section 2.5.3. Note that SJF scheduling can either be implemented in a cooperative or preemptive manner, depending on whether it is desired to interrupt a process with a remaining CPU-burst time that is longer than the interval of the process at the head of the queue (also referred to as *shortest-remaining-time-first*) [112].

In the context of general purpose time sharing systems, *responsiveness* is very important. This raises the need for mechanisms that allow *important* processes (e.g. those that handle UI events) to be scheduled first, such that response time can be minimized. The notion of importance is reflected through priorities (in fact SJF is a special case of priority scheduling [112]). For completeness, this type of scheduling is described briefly. However, for the reasons discussed in section 2.5.3 it is not further considered for the implementation of the *Pony* scheduler.

2.5.2 Priority-based Scheduling

Priority-based scheduling requires some quantifiable means of comparing two processes. This can be implemented through associating an integer variable with each process, or by putting processes in queues that have a priority associated with them. Especially in the context of systems on which rather random types of processes that greatly differ in I/O and CPU intensity are executed, priority-based scheduling proves useful [112]. The main idea is that the process of the highest priority is allocated to the CPU. As a consequence, processes of low priority might wait indefinetly if there is always a process of a higher priority to execute. This is usually avoided by increasing the priority of long-waiting processes such that they eventually get scheduled (also called *aging* [112]). How long a process waits is expressed in terms of time quanta it has not been scheduled.

Multilevel feedback queuing is a fairly complex and preemptive implementation of a scheduling mechanism of this class, that aims to efficiently schedule workloads that are entirely unpredictable. I/O-bound and interactive processes are appended to queues of high priority and others to queues of low priority. The amount of available process queues is implementation dependent. Aging may cause processes to diffuse from lower queues to higher ones to avoid starvation [112]. Some implementations use different scheduling algorithms for queues of different priority (e.g. FCFS for the queue of lowest priority) [112]. The intention is to use the optimal scheduling algorithm per type of process. A process that is committed to a queue preempts a process that is currently running but came from a queue of a lower priority. The result is a very flexible scheduling mechanism. However, flexibility comes with the cost of being forced to find the right parameters for a specific system using this scheme in order to perform as intended (principally because some systems are not as unpredictable as they seem to be).

Priority-based scheduling is unnecessary overhead in the context of a language runtime. Although we must support a broad set of generic applications, the set of entities on which a runtime scheduler operates upon is (at least to some extent) homogeneous.

2.5.3 Operating System Scheduling vs. Actor Runtime Scheduling

There are various differences in the requirements and characteristics between operating system scheduling and language runtime scheduling. While many ideas are similar or even the same, in the context of a language runtime such as *Pony* we can implement a scheduler that is more specific and fit for purpose, rather than providing a generic scheduler for any kind of processes.

The design of process schedulers for time sharing systems was mainly influenced by the fact that these systems are used by people, such that minimizing response time becomes most important due to usability concerns. Additionally, the kind of processes that are running on these systems is in most cases unknown. The goal of minimizing the average response time of processes resulted in the implementation of fairly complex, preemptive and priority-based mechanisms.

The *Pony* scheduler can be far more tailored to specific characteristics of the language. However, the basics of operating system scheduling also apply in this context. The goal is to reduce the turnaround time of an actor as well as to maximize CPU utilization. Figure 2.2 shows a similar state machine as presented in section 2.5.1, in order to illustrate when scheduling has to happen in the *Pony* runtime.

Evidently, the situation looks similar to the one discussed for processes. Upon actor creation, an actor is *blocked*. Blocked actors are never scheduled, because there is no work to do. If a message is put into an actors message queue which was previously empty, the actor becomes *ready*.
Ready actors that are at the head of the scheduling queue are allocated to the CPU for processing

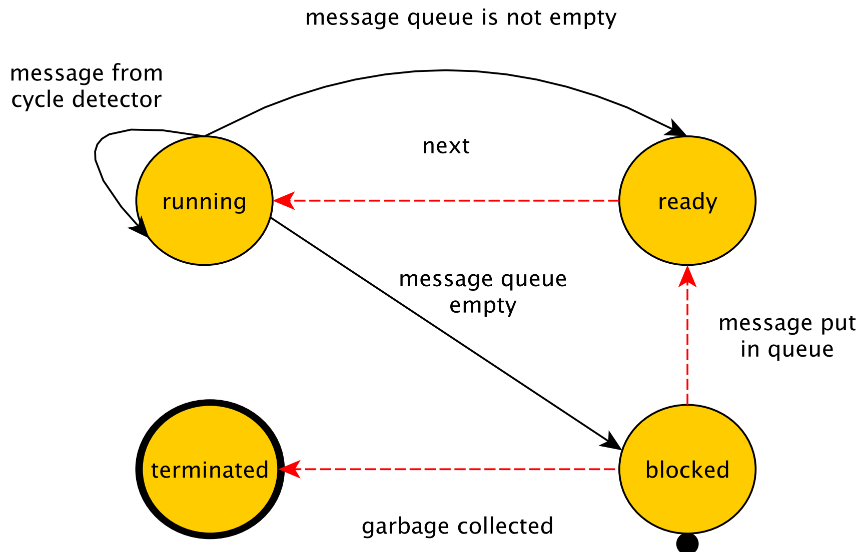


Figure 2.2: State Transitions of Pony Actors

messages. An actor could technically always have messages to process, thus for avoiding starvation we allow an actor to exactly process one message and then schedule the next ready actor. For reasons of computational progress, we allow an actor to process two messages if the next message to process is runtime control message (which are described in chapter 3). If the message processed was the last message in the queue, the actor becomes blocked again.

Due to the nature of the actor programming model and due to appropriate programming practices, dispatching a message should not be a long running CPU-intensive task (and definitely should not include an infinite loop). This means, we are not forced to implement a complex timing mechanism that allocates a fixed quantum to each scheduling slot of an actor. Furthermore, the CPU intensity of processing a single message should not vary greatly between actors. I/O events are handled asynchronously (more detail in chapter 3), such that an actor does not wait for I/O completion.

Consequently, we are not facing the problem that lead to the idea of *shortest-job-first* scheduling as discussed above. An actor that has finished processing a message is put back to the tail of the queue. Therefore, the *Pony* scheduler is (to some extent) preemptive, where the time quantum for each actor is processing a single (application-) message. This is similar to the *Round-Robin* mechanism used in early operating systems [112].

Scalability is not addressed by the above, because it does not attempt to uniformly distribute workload among all available processor cores to maximize CPU utilization. For this reason, each processor core in the system is associated with its own task queue. Each queue is owned by a separate thread. Hence, the runtime forks as many threads as there are cores in the system. The actual *Pony* scheduler is a mechanism, where the above algorithm is applied to each task queue and additionally uses work stealing such that cores are used productively.

In the following, the basic concepts of work stealing are discussed. Technical details on how both mechanisms can be combined to a single efficient scheduler implementation are given in section 2.6. The actor state diagram will be reconsidered for the purpose of the implementation of a distributed scheduler, which is presented in chapter 3.

2.5.4 Work Stealing

Work stealing is a passive (it only runs when it needs to), usually decentralized and dynamic scheduling mechanism where idle nodes/processes steal queued work from busy nodes. This allows for an implementation of a dynamic and adaptive scheduler which does not require prior knowledge

of task dependencies (which makes it inherently good for irregular and a broad set of applications). Furthermore, it is inherently distributed and is therefore well suited for a highly-scalable implementation. It is surely possible to implement a centralized work stealing mechanism by employing a central load balancing server, but in the context of a distributed language runtime we want to avoid network configuration and single points of failures (if possible).

Work stealing algorithms have been implemented in many variations in languages like Cilk [48] and X10 [104]. The differences between languages that implement work stealing mechanisms are usually around the degree of flexibility that is given to the programmer when expressing dependencies between tasks. Processors or nodes that have an empty task queue are usually referred to as "*thieves*" [15] which will attempt to steal work from another busy node's queue ("*victims*" [15]).

From a global perspective, work stealing does not necessarily need to be optimal, as it does not guarantee to find the perfect schedule, which may result in (possibly) slow work distribution. This concern is important for real time systems, but for a language runtime scalability is one of the most important concerns – and not necessarily the optimal schedule. Furthermore, it might not be optimal from an applications point of view if all concurrent entities are uniformly distributed over a network – as computation steps may be extremely quick, such that migration overhead would be too expensive. However, the execution time of work stealing based schedulers as well as the number of stealing requests is theoretically bounded [106].

How (and when) work is stolen is an optimization problem, especially in the context of a programming language runtime, where the set of executed applications is generic and all work may be concentrated on a few nodes. Blumofe et al prove in [15] that randomly selecting a node to steal from can be an efficient choice. However, as discussed later in the context of distributed systems (and depending on the network topology), considering data locality when deciding from which node to steal might be a good optimization [1]. Dependencies between tasks are usually maintained as a (distributed) dependency graph. This information is vital for implementing an algorithm that yields as least work stealing requests as possible in order to achieve a good workload distribution. However, the need for dependencies between tasks is dependent on the type of structure the scheduler is operating on. As actors are essentially self-contained, we do not need to worry about task dependencies as such. More important for a distributed version of *Pony* is to carefully decide whether an actor should be migrated to a remote node or not. Similarly, it might be worth migrating an actor on purpose, if most messages are received from an actor that is located on a remote node. Scheduling tasks with work stealing is provably optimal [14] (within a constant factor) for fully strict parallel computations, where all outgoing edges in the data dependency graph of a thread go to an ancestor [12]. This is clearly the case in *Pony* (section 2.6.4)

Work Stealing Queues

The performance of a work stealing algorithm strongly depends on the underlying queue implementation in which computation tasks are stored [5]. Access to the queue itself is in the critical path of the scheduling mechanism, so it is advisable to implement these queues with low cost synchronization between threads. We cannot predict the level of multiprogramming that might be incorporated in applications developed based on the *Pony* runtime. As a consequence, we do want to avoid expensive queue overflow mechanisms [60] and therefore the work stealing queue of *Pony* is of dynamic size (rather than fixed size as proposed in [5]). The work stealing queue implementation of *Pony* is discussed in more detail in section 2.6.3.

A common approach on when to steal an item from another's node dequeue is when the local dequeue is empty [15]. This also means that only the owning runtime-thread of a queue can add tasks to its queue but all other threads can remove tasks from any queue (usually reflected through *put*, *take* and *steal* operations [101]). The *emptiness* of a queue only signals that we have to steal work from somewhere (otherwise the node would stay idle). In a concurrent setting, the selection of a victim may seem trivial, because the access to any other core's memory bank or cache is uniform from the perspective of a thief. However, the consideration of cache coherency protocols and systems with multiple sockets (not only multiple cores) can make it more complex. In early stages of concurrent *Pony*, randomly selecting a victim was sufficient. However, this changes as

soon as support for distributed computing is introduced, where network communication and the topological distance between nodes may be taken into account.

Hierarchical Work Stealing

Hierarchical work stealing tries to avoid unnecessarily high communication costs caused by long distance communications by taking data locality into account [120].

Data locality for work stealing in hierarchical implementations is reflected through maintaining multiple levels of hierarchy. A prominent example of such a mechanism is called CRS (*Cluster-aware Random Stealing*) [120], which maintains clusters of processors local to a node and processors of remote nodes. Worker queues can be accessed concurrently in local clusters. A processor is then able to send either asynchronous or synchronous stealing messages [120]. Asynchronous messages are point-to-point such that they are restricted to one computer in another cluster. Synchronous requests cause the thief to block and are restricted to nodes in the same cluster. As previously, stealing requests are only sent if a worker's queue is empty [120]. Note that one asynchronous steal request and one synchronous steal can be sent out at the same time.

The main idea of hierarchical work stealing is to first attempt to steal work from nodes physically closest to the thief node. Only if this request fails, requests are made to remote nodes in order to reduce latency of work distribution. The effect of having a network of *Ponies* to be managed based on a k-Tree topology is that the work stealing mechanism implemented for this project is inherently hierarchical (see chapter 3).

Idempotent Work Stealing

As previously mentioned, the performance of work stealing largely depends on the underlying dequeue implementation. Idempotent work stealing [101] is an optimization that allows a more flexible approach. Conventional work stealing semantics guarantee that a task that is submitted to a worker queue is executed exactly once. The motivation for idempotent work stealing is that a wide class of applications allows for weaker guarantees, because either applications check that work is not repeated or it can be tolerated (and are therefore idempotent operations) [101]. Thus, guarantees are only given that a submitted task is executed *at-least-once* [101]. These relaxed semantics allow for a more flexible implementation of a work stealing mechanism. *Exactly-once* semantics require the use of atomic instructions (or memory ordering fence instructions) [28, 61, 48] in the critical path of a dequeues owner-thread. These instructions are substantially slower than regular memory instructions [101].

Note that the relaxation of the mechanisms semantics does not mean that the implementation becomes trivial. It is still necessary to ensure that tasks are not lost and that consistent information is kept without the use of expensive synchronization instructions in the corresponding put and take operations of a dequeue [101]. A family of algorithms for mechanisms based on this paradigm is discussed in [101].

For the implementation of *Pony* we will not consider relaxed work stealing semantics, principally because we do not want the programmer to ensure that implemented actors handle messages in an idempotent manner, which would have a negative effect on the code structure due to unnecessary checks.

2.6 Pony

Prior to the start of this project, work has been carried out to implement a runtime for *Pony* that supports highly efficient concurrent programming. The existing language runtime is the basis for the implementation efforts carried out in this project. In order to give a better understanding of what the current implementation provides and what has to be extended/adpated to allow for scalable distributed programming, various parts of the existing code base are discussed. Note that we are only focusing on the runtime itself, and not – except for the actor programming model – on work that has been done on the language level (e.g. the type system etc.). Figure 2.3 shows

the structure of the runtime system in terms of the components involved and their relationship to each other. Details such as the pool allocator for memory management or components for garbage collection are omitted in this diagram.

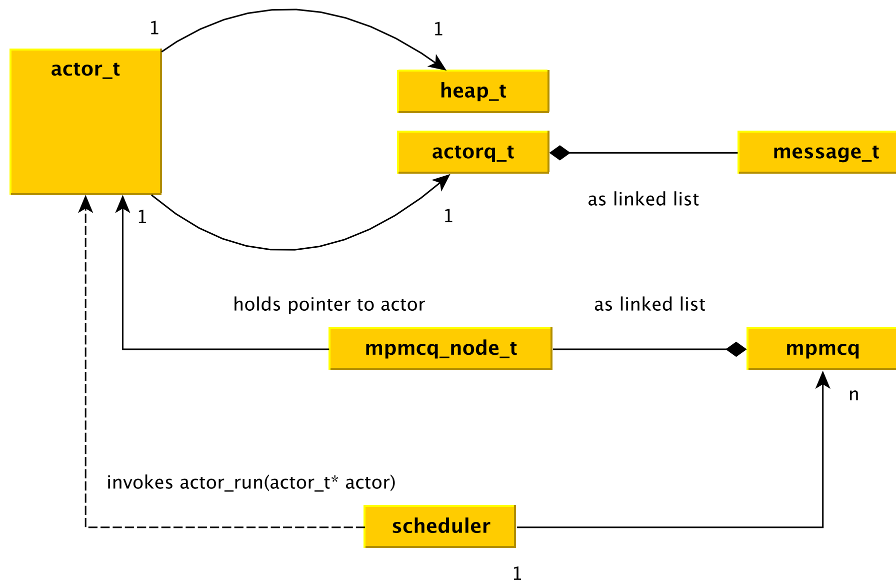


Figure 2.3: Basic structure of the Pony runtime (some details omitted)

Each actor is associated with exactly one heap for private data as well as one local message queue (*actorq_t*) to buffer incoming messages from other actors. A message queue is a concurrent and wait-free linked list, were each node holds a pointer to a structure of type *message_t*, containing a message identifier, the arguments provided for the message as well as a pointer to the next element in the queue.

Upon runtime initialization, the scheduler forks as many threads as there are cores available, where each thread is pinned to one of the available cores (uniformly; also called CPU affinity). Each thread maintains a multiple producer, multiple consumer queue (as linked list), holding actors to be scheduled. This queue is the central data structure for *Pony's* work stealing mechanism. Each node holds a pointer to one actor. Each of the threads (including the main scheduler thread) pop one element at a time from the queue to execute an actor on the core the queue is associated with. If no actor can be scheduled, the corresponding thread tries to steal one element from a randomly picked scheduling queue. If the stealing request cannot be satisfied (see section 2.6.3), the thread gives up its quantum and tries again. The main components are discussed in more detail in the remainder of this section.

2.6.1 The Pony Actor

In *Pony*, actors are the central entity of execution. Listing 2.1 shows the external interface of the actor implementation, providing functions for creating and destroying an actor structure as well as for sending messages and running the actor.

Listing 2.1: External Interface of the Pony Actor

```
1 bool actor_run(actor_t* actor);
2 void actor_destroy(actor_t* actor, map_t* cycle);
3 void actor_sendv(actor_t* to, uint64_t id, int argc, arg_t* argv);
4 actor_t* actor_create(actor_type_t* type);
5 void pony_send(actor_t* to, uint64_t id);
6 void pony_sendv(actor_t* to, uint64_t id, int argc, arg_t* argv);
7 /* convenience functions are omitted */
```

The (runtime) programmer is not exposed to the internals of the actor data type (*struct actor_t*) but is instead provided the corresponding opaque type (*actor_t*; similar to the concept of information hiding in object oriented languages). Hiding the details is sensible, because it allows the underlying actor implementation to be changed and, as long as the interface is kept stable, without the need of recompiling applications that depend on it.

The structure *actor_type_t* describes the representation of an actor (Listing 2.2).

Listing 2.2: Actor Descriptor

```
1 typedef const struct actor_type_t
2 {
3   trace_fn trace;
4   message_type_fn message_type;
5   dispatch_fn dispatch;
6 } actor_type_t;
```

The type of an actor is determined by a pointer to a conversion function that describes the kind of messages to which the actor reacts to (Listing 2.4) as well as of a pointer to a dispatch function which is used as message handler. The function of type *trace_fn* will be used as a hook for serialization and deserialization, as discussed in chapter 3. Furthermore, the field `trace` supports *Pony's* trace mechanism which is important for reference counting and garbage collection. For the purpose of this example it shall be disregarded and is therefore a null pointer. This structure is potentially shared among actors of the same type, and is therefore defined as *const*, such that it remains unchanged up to the point the program terminates.

A message type (implemented as *message_type_t*, Listing 2.3) describes the number of arguments that an actor is expecting for a given type of message as well as a set of trace functions for these arguments. The mode (*pony_mode_t*) is used to identify certain properties of the corresponding argument (e.g. actor, primitive or any type information in future versions).

Listing 2.3: Message Types

```
1 typedef const struct message_type_t
2 {
3   int argc;
4   trace_fn trace[PONY_MAX_ARG];
5   pony_mode_t mode[PONY_MAX_ARG];
6 } message_type_t;
```

The type of a message to be sent is identified by an ID (in the example the only message ID is MSG_RESPONSE, Listing 2.4). Note that these message IDs are dependent on the actor structure and the contract it is providing to other actors. In the future, this mapping will be generated by the *Pony*-Compiler.

Listing 2.4: Declare Messages

```
1 enum
2 {
3     MSG_REPSPONSE
4 };
5
6 static message_type_t m_response = {1, {pony_trace64}, {PONY_PRIMITIVE64}};
7
8 static message_type_t message_type(uint64_t id)
9 {
10     switch(id)
11     {
12         case MSG_RESPONSE: return &m_response;
13     }
14     return NULL;
15 };
```

Sending messages is simply done by providing the target actor, the message identifier, the number of arguments and the arguments themselves, combined in a union type *arg_t*. In the concurrent setting, the target actor is determined by its memory address. However, this changes in the context of distributed computing, because the destination might be located on a remote machine.

Listing 2.5: Union Type for Message Arguments

```
1 typedef union
2 {
3   void* p;
4   intptr_t i;
5   double d;
6 } arg_t;
```

A union type is a structure where each element starts at the same address. Hence, only one element can be used at any one time. As a consequence, the size of a union type is the same as the size of its largest element. This allows to provide a convenient way to handle message arguments of different types (and provide a convenient way of sending primitive arguments) without wasting memory space for unused elements within the structure.

As previously mentioned, the dispatch function is the main implementation of the state changes an actor performs depending on incoming messages. Since we implement the runtime system prior to formulating a complete language specification and a compiler, the examples given throughout this report implement the dispatch function manually.

Listing 2.6: Dispatch Handler

```
1 static void dispatch(actor_t* this, void* p, uint64_t id,
2                      int argc, arg_t* argv)
3 {
4     switch(id)
5     {
6         /* handle message according to ID */
7     }
8 }
```

Listing 2.7 shows how actors can be created within the *Pony* runtime. Note that actor creation internally triggers allocating heap space and setting up any structure necessary for maintaining the actor (message queue etc.). After creation, the actor can be used by the scheduler to be allocated to the CPU, by simply calling *actor_run(actor)* in the context of some thread. Once scheduled, the actor pops a message of its mailbox (if any) and processes it according to the arguments provided. Running an actor causes the dispatch function, which was given upon actor creation, to be invoked only if the incoming message is an application message and not a runtime control message.

Listing 2.7: Creating Actors

```
1 static actor_type_t type =
2 {
3   NULL,
4   message_type,
5   dispatch
6 };
7
8 actor_t* actor =  pony_create(&type);
```

Internally, the dispatch function is additionally provided a pointer to the corresponding actor's heap. Memory is not shared, so incoming messages are *shallow*-copied to the target actor's address space.

Shallow copies are possible due to guarantees given by the *Pony* type system, which is outside of the scope of this project. However, it is a good example of the impact a type system can have on efficiency, because only the `arg_t` structures from listing 2.5 need to be copied. This is called (safe)

zero-copy messaging, because the actual message arguments (except primitives) are not copied, only pointers to them, the type system guarantees the absence of data races.

If the message queue was previously empty (before the incoming message was copied), the actor is added to the tail of the scheduling queue owned by the thread which is currently active (i.e. to the queue that belongs to the core on which the current actor is running on). We do not have to care about finding the queue with the least elements, because the actor will eventually be stolen by some other thread.
As a consequence of the message-passing paradigm, message queues are required to buffer any incoming messages from other actors, which may come in concurrently. Since computational progress is expressed through the sending, receiving and processing of messages, the message queue is in the critical path of an application and must therefore be implemented efficiently.

2.6.2 Message Queues

The underlying data structure of *Pony* message queues is a wait-free, multiple producer and single consumer linked list (FIFO, `actorq.c`). The queue needs to support multiple producers, because there could be as many actors as available cores trying to send a message to the same actor concurrently. However, there is only one thread at a time that takes elements from the list, because actors never consume messages from queues other than their own.

This dictates the design of the queue: the operation that adds an element to the head of the list (`actorq_push(...)`) must be *thread-safe*, whereas removing a message (`actorq_pop(...)`) can simply pop the tail of the list without any thread synchronization if the emptiness of the queue is handled properly.

The structure of a linked list is trivial. We only need to maintain a pointer to the head and to the tail of the list. In the case of *Pony*, the elements are of type *message_t* (Listing 2.8). The head of the queue is marked as *volatile*, because it may be modified by many threads. The tail however is only modified by one thread at a time. Each element maintains a pointer to the next element of the list. The ID of a message is equivalent to the one discussed previously. The modes of the arguments and their trace functions do not need to be part of the message structure itself, because they can be retrieved via the message conversion function provided by the type of the receiving actor.

Listing 2.8: Structure of Message Queue

```
1 typedef struct message_t
2 {
3   uint64_t id;
4   arg_t argv[PONY_MAX_ARG];
5   volatile struct message_t* next;
6 } message_t;
7
8 typedef struct actorq_t
9 {
10   volatile message_t* head;
11   message_t* tail;
12 } actorq_t;
```

The challenge in the list implementation is to find an efficient way to signal the emptiness of a list as well as the case when there is only one element available (i.e. tail and head point to the same message). It is especially important to think about the latter, because there is a race condition between adding an element to a list that currently has one element and removing the only element. However, we want to leverage the special single consumer characteristic of an actor's mailbox.

The solution implemented is closely connected to *Pony's* pool allocator (section 2.6.5). A pool for a specific type is always aligned up to the L1 cache line size (on today's Intel processors, 64B). This guarantees that the first five bit of a pool's base address are always zero. As a result, we are able to signal the emptiness of the list with a stub element that is identified by the first bit of its address being set to one (because this bit doesn't affect the semantic integrity of the pointer if

handled carefully).

Upon initialization of the list we allocate a new pool. Since the list is empty at the beginning, we mark the first element as stub element. Listing 2.9 shows that this can be done by letting the head of the list point to the newly allocated stub element with a one-bit offset. The tail points to the actual stub base-address (`|` is the logical bit-wise OR operator). Note that the stub's next pointer is `NULL`.

Listing 2.9: Initialisation of Message Queue

```
1 void actorq_init(actorq_t* q)
2 {
3   message_t* stub = POOL_ALLOC(message_t);
4
5   q->head = (message_t*)((uintptr_t)stub | 1);
6   q->tail = stub;
7 }
```

This allows to test for emptiness by checking whether the expression `((uintptr_t)elem & 1))` equals zero (in which case the list is non-empty) or not. The important thing is to be cautious when adding a new element, because the structure pointed to by *head* is now displaced by one bit. Thus, we are required to undo the above operation when inserting a new element. It is not problematic to reset the last bit whenever a new element is added, since there cannot be two *message_t* elements within a two-bit address boundary.

Listing 2.10 shows that adding an element is implemented with the compiler built-in operation `__sync_lock_test_and_set` [52] that atomically assigns the value of `msg` to the variable that `q->head` points to (by implicitly creating a memory barrier). The return value is a pointer to the memory location where `q->head` had pointed to before it was exchanged. A *lock* or *mutex* is never acquired. The operation `actorq_push` returns whether the list was empty or not in order to unblock the actor if necessary.

Listing 2.10: Adding an element to the list

```
1 bool actorq_push(actorq_t* q, message_t* msg)
2 {
3   msg->next = NULL;
4   message_t* prev = (message_t*)__sync_lock_test_and_set(&q->head, msg);
5
6   bool was_empty = ((uintptr_t)prev & 1) != 0;
7   prev = (message_t*)((uintptr_t)prev & ~(uintptr_t)1);
8
9   prev->next = msg;
10
11  return was_empty;
12 }
```

The benefit of the above is that this scheme allows to remove an element from the tail of the list without using an atomic operation or any synchronization mechanism that caters for the race condition discussed earlier. Hence, an actor consuming a message from its mailbox is not impeded by other actors that send messages to it.

Remember that after initialization, the tail pointed to the stub element, and the stub's next pointer was `NULL`. The operation that takes a message from the queue (Listing 2.11) can safely return `q->tail->next` in any case, because `actorq_push` does not remove the stub. Returning `q->tail->next` instead of `q->tail` is the pinpoint of why consuming a message can be done without any synchronization. Moving the tail to the former successor is valid at any time, because this pointer is not modified by any of the threads that might append a message. The tail pointer continuously follows the head pointer. A nice side-effect of this approach is that the returned list item is deallocated after the next message was successfully retrieved and when it is returned. This is required, because we need to ensure that the actor has processed the message before it is deallocated, which must have happened when the actor comes back and probes for new messages. By doing so, managing the memory for the message queue is concealed within the `actorq_t` implementation.

Listing 2.11: Removing an element from the list

```
1 message_t* actorq_pop(actorq_t* q)
2 {
3   message_t* tail = q->tail;
4   message_t* next = (message_t*)tail->next;
5 
6   if(next != NULL)
7   {
8     q->tail = next;
9     POOL_FREE(message_t, tail);
10  }
11 
12  return next;
13 }
```

The only problem that remains is that the tail pointer would be forwarded to the position where the current head pointer points to if the element returned was the last in the list. This element needs to be remarked as stub, because testing for emptiness was important for blocking and unblocking an actor. There is a potential race condition between remarking the element as stub and adding a new message to the queue, which can be eliminated efficiently using `__sync_bool_compare_and_swap` [51]. This operation atomically exchanges a value by another if it is equal to a third. The return value is boolean and indicates whether the values were equal and the atomic exchange was therefore executed, or they were not equal and no operation was done.

A linked list is also the central data structure for the work stealing scheduler. Contrary to the message queue described above, scheduling queues have to support multiple consumers, which makes the implementation considerably more complex.

2.6.3 Work Stealing Queue

The data structure on which the work stealing queue (`mpmcq.c`) of each scheduler thread depends is critical and needs to provide efficient push and pop operations, because the access to the queues is on the critical path of assigning actors to cores and execute them.

The message queue discussed previously already supported multiple producers, so the implementation for adding a new node to any of the scheduling queues is almost identical (only the stub is handled slightly differently). The main challenge of multiple consumer queues is the *ABA* problem [99]. Consider the following example:

- Assume a singly-linked list q (FIFO) contains the following elements: `A -> B -> C`, where A is at the head of the queue and C at the tail. Consuming a value deallocates the element.
- Two concurrent threads T_1 and T_2 access the list concurrently, trying to consume a value.
- T_1 reads the tail of the queue and determines the predecessor. Assume the following assignments are performed: `node* t = C; node* prev = B`. After doing so, T_1 becomes preempted.
- T_2 becomes scheduled and consumes a value. The result is `q = A -> B`. After that, thread T_2 may pop another element, which leaves `q = A`.
- Another thread T_3 may concurrently produce a value and because we are not using a garbage collected language (for implementing the runtime), this value may very well be allocated at the memory address that C had when T_1 tried to consume it. We have `q = A -> C`.
- When resumed, T_1 checks whether the tail of the list was concurrently changed and if not, replaces the tail of the list with `B` (by using `compare_and_swap`). The atomic instruction would succeed (because `t = C`), although the state changed in between. As a result, thread T_1 causes the new tail of the list to point to `B` which has been previously deallocated by T_2.

  This is obviously a memory violation.

In the programming language C, accessing freed memory results in undefined behavior, and will likely cause the program to crash. If portability between 64-bit and 32-bit machines is important, this problem can be solved using *hazard pointers* [100]. However, almost any of today's processors support 64-bit addressing, such that *Pony* can make use of a lightweight mechanism using *tagging* and *double-word compare-and-swap*, proposed by IBM in 2003 [98]. As a result, the tail node of *Pony's* work stealing queue is defined to support double-word operations (Listing 2.12, alignments are omitted).

Listing 2.12: mpmcq double-word type

```
1  typedef struct mpmcq_dwcas_t
2  {
3    union
4    {
5      struct
6      {
7        uint64_t aba;
8        mpmcq_node_t* node;
9      };
10
11     __int128_t dw;
12   };
13 } mpmcq_dwcas_t;
```

The idea is to associate a tag with a node and recheck that tag whenever it might have been changed concurrently. The union type allows that both the tag and the memory address (which are 8 byte integers on 64-bit machines) of the actual node can be given to the atomic compare-and-swap operation as shown in Listing 2.13.

Listing 2.13: Removing an element ABA-safe

```
1  void* mpmcq_pop(mpmcq_t* q)
2  {
3    mpmcq_dwcas_t cmp, xchg;
4    mpmcq_node_t* next;
5    void* data;
6
7    do
8    {
9      cmp = q->tail;
10     next = (mpmcq_node_t*)cmp.node->next;
11
12     if(next == NULL)
13     {
14       return NULL;
15     }
16
17     data = next->data;
18     xchg.node = next;
19     xchg.aba = cmp.aba + 1;
20   } while(!__sync_bool_compare_and_swap(&q->tail.dw, cmp.dw, xchg.dw));
21
22   mpmcq_free(cmp.node);
23   return data;
24 }
```

"Versioning" the tail nodes ensures that an element, once touched, can be uniquely identified and compare-and-swap does the rest of the trick. The tail pointer is exchanged implicitly because the union type guarantees that the memory address of the node is overwritten accordingly. If two threads are fighting for the same element, one of the two will fail in completing the compare-and-swap operation and therefore tries again. The same applies for ABA-detection. Contrary to the message queue implementation described above, signaling the emptiness of the scheduler queue is

not of importance in this context, because the only thing that matters is whether we are able to retrieve a value or not. If the list is empty, attempting to pop an element simply returns `NULL`.

The part of the scheduler responsible for work stealing makes use of the same pop operation. For efficiency reasons (e.g. cache locality) it would be undesirable to allow that a queue is stolen empty. Moreover, we need to avoid in the case of all other queues being empty, that the last actor is "travelling" between queues continuously. Before extending *Pony* with distributed scheduling, we will briefly discuss the design of the single-host scheduler.

2.6.4 The Scheduler

Pony's scheduler is responsible of setting up, managing and assigning necessary threads to CPU cores appropriately. Each scheduling queue is managed in a round-robin manner, with additional work stealing capabilities.

The runtime system (and therefore the scheduler) is started with a call to the function `pony_start (int argc, char** argv, actor_t* actor)`, which is provided the command line arguments and the main actor as entry point for a given application. By default, as many threads as there are physical cores available are created. This can be changed by using the command line option `--ponythreads <n>`, which might result in logical cores being used as well. Determining the number of threads to be forked and pinning them to physical or logical cores appropriately requires to retrieve the processor's properties from the operating system. On linux-based machines, the logical core count can be retrieved via a system call to `sysconf` and providing the parameter `_SC_NPROCESSORS_ONLN` [45]. Note that the result is not necessarily equal to the actual logical core count, because the operating system may decide to turn off cores independently, and therefore this call returns the number of cores that are currently online. The exact logical core count can be retrieved with the parameter `_SC_NPROCESSORS_CONF` [45]. Deciding whether a core is physical or logical is more complex on linux systems, because it requires to read the CPU topology from the file system. MacOSX systems provide the same information through the system properties `hw.physicalcpu` and `hw.logicalcpu`. Letting the operating system schedule a thread on a specific core can be configured using the system calls `sched_setaffinity` [91] on linux systems and `thread_policy_set` [4] on MacOSX systems.

The thread library in use is based on `pthread` [66], a standard specification for creating and manipulating threads. Upon creation, each thread is given a pointer to a function to execute (`run_thread`, implemented in `scheduler.c`). This function is the main scheduling loop. The idea is to implement an infinite loop where each thread tries to retrieve an actor to execute from its own scheduling queue or steals work from any of the others. Work stealing is only performed when a thread's local queue is empty. Once an actor is scheduled, it is given a quantum of handling exactly one application message. For the purpose of computational progress, an actor is not preempted after having processed a runtime control message. This hides the effect of runtime management on application execution.

For efficiency reasons, actor queues are prevented from being stolen empty. This does not mean that another thread is unable to work steal the (currently) last actor in the queue, while the owning thread is executing an actor. In this case, stealing the currently last actor is fine. However, we want to avoid that when a queue is empty, and the currently executed actor does not become blocked (and therefore has more messages to process), that this actor is stolen by another thread. This would be inefficient due to cache locality. Hence, if a thread's scheduling queue is empty when trying to execute the next actor and the previously executed actor did not become blocked, then we simply do not add it back to the queue and continue executing it. This is safe, because an actor is re-added to some queue when becoming unblocked again.

The scheduler threads exit and can be joined using a termination scheme based on quiescence, as described in section 2.6.7.

2.6.5 Pool Allocator

Implementing a memory allocation mechanism is complex and has a large influence on the performance of an application. Generic heap allocators such as *malloc* [79] need to satisfy a large number of different features. As a consequence, they are required to trade-off competing factors against each other, some of which are re-using memory appropriately (which can cause the entire heap to be scanned), speed of deallocating memory and merging blocks of freed memory to avoid fragmentation. Merging free blocks causes that providing any bounds on the time required for deallocating or allocating memory is not possible. This is problematic, because the performance behavior is dependent on the workload of a given application, which can obviously largely differ in the context of a programming language runtime and is practically unpredictable.

Since *Pony* is garbage collected and the programmer is not supposed to explicitly allocate memory, we can provide an allocation mechanism that is less generic. This allows us to tune it more aggressively. *Pony's* allocation scheme (`pool.c`) partitions the memory into pools of the same type (i.e. structures of type `message_t` are always allocated to the same pool). Hence, finding free memory space and deallocating memory is possible in constant time and the problem of fragmentation is avoided implicitly.

A pool for a specific type is always cache-line size aligned (on Intel platforms usually 64 B). This ensures aligned memory access and good cache utilization. A pool of type t needs to be initialized with `POOL_CREATE(t)`. Retrieving a pointer to a memory region that allows to store a structure of the corresponding type t can be done using `POOL_ALLOC(t)`. Pools are essentially managed in a linked list (similar to the scheduling queue discussed earlier). If no current pool can be retrieved, a new pool of 64 KB is allocated using *mmap* [90]. The system call *mmap* is an efficient choice for *Pony's* pool allocator, because upon invocation it just maps the requested memory size to the virtual address space of the runtime process. The necessary operating system work is done as soon as some thread first writes to the pool. This avoids expensive work to be done upon each allocation. Instead, the operating system management only comes into play once for each 64 KB pool. At the same time, *mmap* guarantees that the requested virtual memory is page-aligned. Not only is providing a pool allocator advantageous from the perspective of efficiency, but also from a software engineering view. Memory management is encapsulated within the pool implementation. This contributes to better debugging of memory-related bugs and provides confidence that the runtime system is free of memory leaks.

Note that there is a difference between the runtime system managing memory internally (using `POOL_ALLOC`) and heap allocation for actors, which is based on `pony_alloc`.

2.6.6 Garbage Collection

Available actor-based language runtimes either require the programmer to explicitly manage an actors lifetime or implement garbage collection mechanisms which require thread synchronization, preventing them from being fully concurrent. *Pony* however implements a fully concurrent garbage collector which itself is based on message passing *and* is able to detect cycles between blocked actors in a graph of actors (which itself may mutate concurrently).

Therefore, *Pony* provides a cycle detector, uses deferred direct reference counting and provides a confirmation protocol to deal with concurrent mutation of the actor graph. Deferred direct reference counting is a scheme that overestimates the set of outgoing references (also called *external set*) in order to allow for lazy reference counting (the external set may differ from an actors heap). Garbage collection of actors is problematic, because it is difficult to observe the global state of an actor-based application.

Pony's garbage collection protocol is fully based on the message passing paradigm of the actor programming model. Whenever an actor receives a reference to another actor or object within a message, it adds this reference to its external set. Once an actor performs a local garbage collection cycle, an actor's external set is eventually compacted. References removed from the set are dropped and cause a decrement message to be sent to the corresponding actor or to the actor owning the dropped object. Due to causal delivery of messages in *Concurrent Pony*, a blocked actor with a

reference count of zero is unreachable and can therefore be collected. Moreover, an actor is said to be *dead*, if it is blocked and all actors that reference it are blocked, transitively [30].

The main challenge that garbage collection systems face is that of cyclic garbage, which cannot be collected using reference counting alone. The *cycle detector* (which itself is implemented as actor) keeps track of the actor topology (ingoing and outgoing references of every actor in the system) and detects cycles between blocked actors. Whenever an actor becomes blocked (when its message queue is empty), it sends its view of the topology to the cycle detector. An actor's topology consists of its reference count (ingoing graph edges) as well as a set of potentially reachable actors (the external set). An actor becoming unblocked informs the cycle detector by sending it an unblock message, invalidating the view on the topology of the sending actor. Combining all *blocked* messages from actors in the system can be used to detect cycles between them. The challenge is that the topology of a specific actor is not available directly, but distributed across many actors in the system. Additionally, the topology does not only change when the actor itself mutates but also when other actors mutate [30]. As a consequence, the view an actor has on its topology as well as the view the cycle detector may have on an actors topology can be out of sync and there is no way to track the true topology efficiently [30], because actors cannot directly mutate the reference count of actors they reference. Hence, every actor's view on the topology (including that of the cycle detector) is only *eventually* consistent. Note that it would be possible to let actors directly perform cycle detection just before blocking. However, for this to work would require every actor to maintain a view of every other actors topology and also send block and unblock messages to all other actors. This is obviously inefficient from both the perspective of the time and space complexity. Introducing one level of indirection by using a central cycle detector reduces the message complexity to exactly one unblock or block message per actor [30].

Any cycle which is not confirmed is therefore called a *perceived* cycle and must not be collected immediately, because it is first necessary to ensure that the perceived cycle agrees with the true topology. This problem is solved with a confirmation protocol that allows to check the validity of the cycle detector's view without any synchronization or heap inspection of actors being necessary [30].

The cycle detector can determine whether a perceived cycle agrees with the true topology (and if so is called a *true cycle* [30]), by sending a confirmation message to every actor in the cycle. This message contains a unique token that identifies the cycle in question. Upon receipt of such a message, every actor responds with an acknowledgment, regardless of the view on its own topology. The beauty of *Pony's* cycle detection mechanism comes from guaranteed causal delivery of messages. If the cycle detector does not receive an unblock message before receiving the acknowledgment, then the decision based on which the cycle was detected must have been a topology that agrees with that of the confirming actor (because it did not unblock in between). If all actors in the perceived cycle can be confirmed, a true cycle is detected and all participating actors can be collected. However, if one of the actors unblocks before sending a confirmation message, the cycle detector needs to cancel all perceived cycles of which the unconfirmed actor is part of.

Since we intend to use *Pony's* garbage collector in the distributed context, *causality* must also be guaranteed when sending messages between nodes in a network of *Ponies*. We will bring this topic back to attention in chapter 3. Having provided a fully concurrent garbage collection mechanism releases the programmer of thinking about the lifetime of actors (besides increasing efficiency). This enables the runtime system to automatically terminate an application.

2.6.7 Termination

Contrary to other implementations of the actor programming model, such as *Scala* [40], *Pony* does not require the programmer to explicitly terminate all actors participating in an application. This requires language level support and a runtime check to distinguish between live and terminated actors [30]. In *Pony*, termination is based upon the characteristics of actor-based applications that computational progress can only happen as long as actors send messages.

The idea is to terminate a program based on quiescence. For this purpose, the scheduler maintains knowledge about threads having signalled that they failed in stealing from any of the other

work stealing queues. If every scheduler thread has reported that there is no actor available to be executed (by atomically incrementing a counter), there can not be any messages left. In this case, all available actors must be blocked. Hence, we can safely join all scheduler threads and terminate the application.

We have to reassess this approach for the distributed context, especially because it interferes with the scheduling of network events. The problem is that there might always be a message to be delivered from a remote node. Section 3.8 provides an extended termination algorithm to address this issue.

2.7 Available Actor Languages and Runtimes

There are other actor-based runtimes available, each of which have different characteristics and attempt to be optimized for different purposes. *Erlang* [38] is a functional language based on the actor paradigm, where actors are called *processes*. The main focus of *Erlang* is the development of concurrent and failure tolerant distributed applications. Contrary to *Distributed Pony*, developed for this project, *Erlang* requires to explicitly spawn *processes* on nodes of a distributed system. Furthermore, *processes* cannot migrate to another machine. However, *Erlang* provides built-in fail-over capability, where a *process* which is suspected to have failed can be dynamically exchanged at runtime with a copy of it. Zero-copy-messaging semantics compatible with multi-processing are not available [72].

Scala [40], a JVM-based programming language, does not provide type system support for message-passing. Contrary to *Pony*, *Scala* either spawns a separate thread per actor or executes actors on a thread pool. Both concerns have an impact on *Scala's* efficiency, especially because creating *threads* is a non-trivial task for the operating system and incurs substantial context-switching overhead for scheduling actors. Although Scala supports zero-copy messaging semantics, type system support to avoid data races is not provided. As a result, programmers are required to implement synchronization mechanisms to protect mutable types from being modified concurrently. Therefore, *Scala* combines two different concepts of concurrency. Furthermore, it is not enabled for distributed computing without extensive use of libraries. Using available libraries, distribution is explicit to the programmer and a distributed scheduler is not provided [3].

Similar to *Scala*, *SALSA* [122] is another JVM-based implementation of the actor programming model. *SALSA* allows for actors to migrate, but the programmer is required to explicitly move actors from one host to another. Contrary to *Distributed Pony*, a scheduler to automate this process is not provided. Furthermore, in a concurrent setting, messages are copied independently of the type of the arguments.

There are many other *framework*-based implementations of the actor programming model that do not attempt to provide a programming language as such, but rather a runtime library. Implementations with increasing popularity include *Akka* [16], *Kilim* [115], *Theron* [95] and *libcppa* [26].

Akka provides a JVM-based implementation for concurrent and distributed computing with actors. Moreover, distribution is (to some extent) not exposed to the programmer. However, instead of providing a scheduler for actor migration, distribution and therefore scalability is driven by configuration. This allows to create actors of a certain type on a specific node. As a result, the distribution scheme is static and not reacting to free resources in a cluster.

Both *Theron* (C++) and *Kilim* (Java) are actor implementations for concurrent computing. Contrary to *Scala*, both *Theron* and *Kilim* use a thread pool to execute actor code. The *Theron* scheduler is non-preemptive and based on a simple FIFO queue holding unblocked actors. Work stealing capability is not provided. Contrary to that, *Kilim* provides a user level scheduler that allows a programmer to pick the thread pool an actor should be allocated to. The optimality of such a scheduling scheme is questionable.

Another concurrent runtime written in C++ is *libcppa*, which was designed with distribution in mind. As for any of the implementations above, a distributed scheduler is not provided. Distribution is therefore exposed to the programmer and uses a mechanism similar to that of asynchronous remote procedure calls. The programmer is explicitly required to connect to remote actors, which causes a stub/proxy to be created on both nodes for message delegation.

Note that none of the implementations above provide fully concurrent garbage collection of actors. *Pony*, both in the concurrent and distributed setting, does.

2.8 Conclusions

Designing a programming language is a long and challenging process. The programming model chosen affects various parts of the runtime implementation and dictates the way programmers express their thoughts to develop an application. This is especially true for *Pony*, where the actor programming model as well as causal message delivery enables the runtime system to provide fully concurrent garbage collection of actors. It becomes clear in the remainder of this thesis, that the runtime implementation for *Distributed Pony* also benefits from causal message delivery to a large extent.

Expressing concurrency through asynchronous messaging can beat any form of thread synchronization whilst being expressively equivalent. However, type system support is critical for an efficient message passing system. For example, *Pony* is able to implement zero-copy messaging due to guarantees given by the type system.

The scheduling of actors is important for using the underlying resources to their capacity. Many aspects can be learned from the research field of operating systems for this purpose. *Work stealing* is provably optimal for task and data parallelism [14, 12]. Next to the programming model, the implementation of the scheduler is key for developing a programming language for high performance computing whilst providing high levels of abstraction. This is even more the case in the context of programming distributed systems.

A language based on the actor model is well-suited to be extended for *transparent* distributed programming. Similar to hiding the complexity that comes with explicit concurrent programming, we believe that it should be possible to program actor-based applications uniformly, such that applications scale in a concurrent as well as in a distributed context without changes to the code being necessary.

The challenges of distributed programming are *partial failure* and *causality*. *Guaranteed causality* is not a common property among programming languages. We believe that it is an important tool to aid in reasonability about programs. Although handling *partial failure* is important and may even influence the design of a language (both syntactically and semantically), we first decided to pursue a distributed runtime implementation that guarantees causal message delivery. The semantics of a language runtime scheduler are not only important for the performance of applications, but also for program termination. Since actors are loosely coupled and send asynchronous messages, many actor-based runtimes require that the programmer explicitly manages the lifetime of all actors. *Pony* attempts to release the programmer from this responsibility.

Chapter 3

Distributed Pony

In this chapter, we describe the design choices made for extending the *Pony* runtime for distributed computing and provide a brief motivation, summary of possible alternatives, their analysis and evaluation in order to illustrate the design process carried out for this project.

Distributing computation tasks requires to provide I/O capability for network communication between runtimes. The implementation and scheduling of read and write operations to network sockets and accepting incoming connections is absolutely critical for the overall performance. After having presented an architectural overview in section 3.1, section 3.2 provides a detailed discussion on how I/O events are multiplexed and when these events are handled by the language runtime scheduler of Pony. Interestingly, a characteristic that is intrinsic for the Pony runtime due to the programming model is also an important factor for scheduling I/O events efficiently: the *Distribution Actor*.

Section 3.2.4 explains the underlying network stack implementation based on Berkeley sockets. This part is important for the asynchronous networking capability and its performance as the underlying socket implementation used by the Distribution-Actor should be non-blocking. We designed a framed communication protocol for two types of *Pony* control messages (additionally to application messages): scheduler messages and cycle detector messages. The same protocol is also used to dispatch messages between actors located on different sites.

Delivering messages to remote actors and migrating actors from one node to another requires a serialization and deserialization mechanism (section 3.4). The process of flattening object structures before posting messages to the networking layer or satisfying remote work stealing requests does not directly contribute to computational progress and must therefore be designed carefully and with efficiency in mind. In order to improve reasonability about application performance, serialization and deserialization should not be a dominant factor. Note that the need for such a mechanism may also have an effect on actor migration, as pinning actors to a node if most messages are local to that node (i.e. come from and are delivered to actors on the same node) might be a winning strategy (section 5).

Section 3.5 and 3.7 focus on the adaptation of *Pony's* work stealing scheduler as well as the cycle detector used for fully concurrent garbage collection of actors. As previously mentioned, the cycle detector depends on the causal order of messages. A formal argument that causality is guaranteed in tree network topologies is given in chapter 4.

At the language level, object identity comparison is supported. This is more complex in a distributed setting. A possible implementation of distributed object identity comparison is discussed in section 3.6. We conclude with termination in distributed and actor-based systems in section 3.8.

3.1 Overview of Runtime Components

The following overview – illustrated in Figure 3.1 – provides a summary of the main components which enable *Pony* for distributed computing and gives a discussion about their responsibilities as well as the dependencies between them.

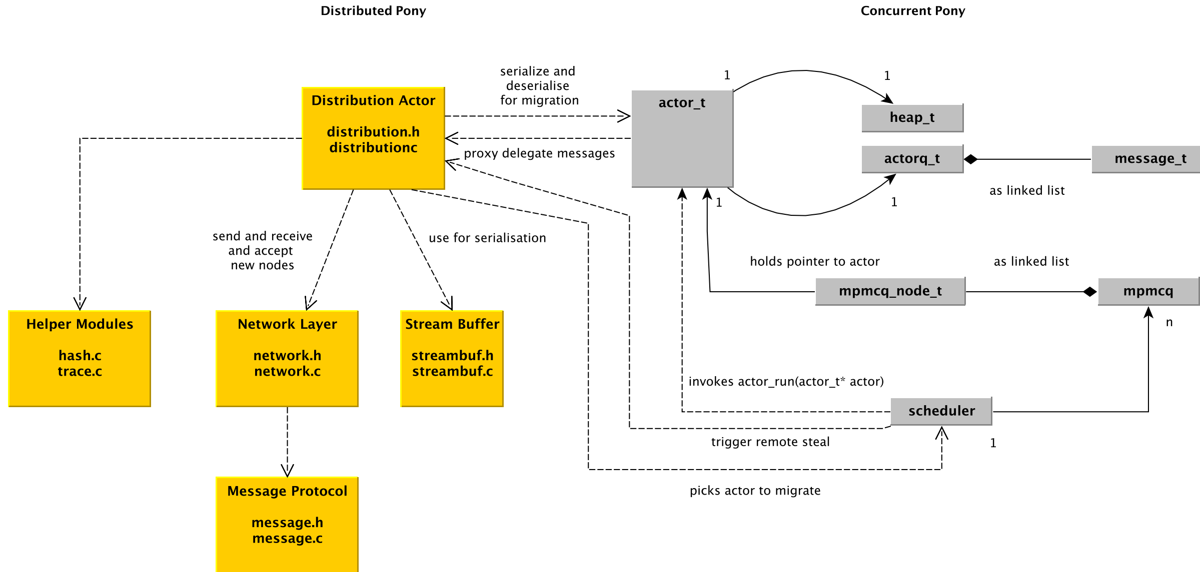


Figure 3.1: Overview of Runtime Components

The main component is the *Distribution Actor* (`distribution.c`, existent on every node participating in a cluster of *Ponies*), which is the only entity (together with the network layer) in the system that explicitly maintains knowledge about the network and any concerns related to distributed scheduling of application actors. This knowledge is never exposed to any other parts of the system. Application actors never communicate with the *Distribution Actor* directly. Instead, they communicate with so called *actor proxies* created by the *Distribution Actor* for scheduling purposes. Proxies are actors that delegate application messages to the *Distribution Actor*, which in turn cares about sending these messages to a remote node for which they are intended. Any data structures required for managing a network of *Ponies* are only accessed by the *Distribution Actor*. Although concurrency is involved, the implementation of *Distributed Pony* is free of any locks for thread synchronisation.

Moreover, the *Distribution Actor* has access to functions implemented by the local scheduler (`scheduler.c`), because we might be required to steal work from any of the local scheduling queues to migrate actors to remote nodes. The implementation of the local scheduler is responsible for triggering the remote work stealing mechanism if there are local cores in idle state.

The *Distribution Actor* depends on the network layer (`network.c`), which implements all low-level primitives for network communication including a non-blocking implementation for sending and receiving data to and from remote nodes as well as a framed network protocol. The network layer manages any topological information, and therefore knows about any directly connected nodes. A global view of the entire network is not maintained. The combination of the network layer and the *Distribution Actor* implements the work stealing heuristics as they are described in section 3.5. Furthermore, the *Distribution Actor* is responsible for routing messages from source to destination as well as for distributed termination, serialization, deserialization and plays an important role for distributed garbage collection.

Sending messages and migrating actors from one node to another requires serialization and deserialization of actors, references and objects. As a consequence, we have implemented a stream buffer (`streambuf.c`) for outbound data. The stream buffer is important for the performance of

serialization as well as for reducing the amount (and therefore overhead) of system calls necessary for network communication. An additional buffer for inbound data is not required, because messages are either stored in the network-socket buffer managed by the operating system or are immediately delegated to some actor's mailbox.

A network of *Pony* runtimes is maintained in a tree network topology. The reasons for that topology are presented in chapter 4. A *Pony* node can either be the *master* (i.e. the root of the tree network) or a *slave* node that is connected to its parent node and holds connections to up to $k \in \mathbb{N}$ children. The *master* node is responsible for invoking the main actor of an application. Any new node attempting to join a *Pony* cluster requests access to the network via the *master* node. In order to ease configuration and setting up a network of *Ponies*, the joining process of *Distributed Pony* delegates the new slave to its parent node and guarantees that the tree topology is eventually balanced (section 3.3.2). As a result, no information about the structure of a *Pony* cluster is required in order to add new nodes. The network topology is important for the implementation of the work stealing heuristics and their data locality.

The following sections discuss the details for each of these components and provide pseudo-code of those parts that are of particular interest.

3.2 Asynchronous I/O Multiplexing

The complexity of implementing high performance networking applications is easily underestimated. Suppose we have a Pony node maintaining several incoming connections from other nodes of the system. Once a message is delivered through any of the connections, the receiving node should react to that message by either copying it to a target actor's message queue (which can also be the cycle detector) or by processing work stealing requests. There is no indication on which node sends a message first and when, such that the mechanism for receiving messages must be generic. The blocking nature of the socket receive function of the BSD Socket API [44, 81, 82, 84, 86] (which is used for any kind of network tasks in the *Distributed Pony* runtime) does not allow for handling requests that may come in on any of the other connections while waiting for incoming data on a specific socket. This is also referred to as the C10K problem [73]. For that reason, asynchronous I/O multiplexing is required.

Blocking on one socket or busy waiting in a loop and checking the sockets for available data is obviously not an option. To avoid that blocking on one socket affects data arrivals on other connections, one possible solution could be to spawn a separate thread for each connection. However, spawning and maintaining threads is a non-trivial and costly operation for the operating system kernel. Furthermore, maintaining a dedicated stack for each connection increases the memory footprint of the runtime system and therefore degrades cache locality. The goal is to efficiently listen on one or multiple sockets without wasting too many CPU resources but also with a short latency between a stream of data having arrived at a socket and it being read and processed. Preferably, we only want to process I/O when there is work to do; otherwise we want to assign the available resources to any other tasks that have to be executed. Thus, we want to have a mechanism that signals the *readiness* of a socket. A socket is said to be *ready* if the corresponding operation on it (i.e. read or write) will not block. This allows for quickly polling the states of available sockets and then perform non-blocking operations on them.

In order to implement this mechanism efficiently, OS kernel support[1] is required. In the following, several system calls that are available are discussed. Various characteristics of these APIs and how I/O events should be scheduled will be used in order to motivate the design choices made for the implementation of *Pony's Distribution Actor*.

[1]The implementation tasks of this project are carried out on UNIX based operating systems

3.2.1 Epoll, KQueue and Kevent

Applications that have to handle a vast amount of connections concurrently are supported by operating systems through various system calls that allow to implement asynchronous I/O efficiently. BSD-based systems such as FreeBSD or MacOSX implement a kernel event notification system called *kqueue* [80]. The Linux equivalent (which is technically inferior compared to *kqueue*) is called *epoll* [88]. Both are effectively based on the work carried out by Banga et al. in [7], with the addition of some improvements. Windows implementations are *I/O completion ports* [33] or asynchronous procedure calls (APCs) [32].

Contrary to *select* and *poll*, which are discussed in the next section, the idea of complex notification systems like the above is to implement *stateful* interest sets. An interest set is a data structure that holds information about which connection should be monitored for which event (e.g. reads or writes). These interest sets are said to be stateful because they are maintained by the operating system kernel, which allows them to be incrementally updated, rather than copied from the user application to the kernel for each system call. This approach is motivated by the observation [7], that the *readiness* of network connections or other kernel objects is often relatively sparse. As a result, *stateless* implementations waste CPU cycles on copying and scanning the entire interest sets for no reason. This is obviously problematic in the context of an application which has to maintain many thousand connections (particularly the need of copying for each subsequent call).

Instead, Banga et al propose a *stateful* protocol, where interest sets can be declared and updated incrementally via the function `declare_interest` [7]. User applications can retrieve signaled events via `get_next_event` [7], which effectively dequeues items from an internally maintained kernel queue. The operating system kernel registers asynchronous event handlers for each connection, such that resources are only used when some event occurs that matches any of the declared interests. As a result, available *stateful* notification systems perform in $O(1)$ as opposed to their *stateless* counterparts, which are of linear complexity. The *epoll* equivalent to the two functions mentioned above is `epoll_ctl` for registering events and `epoll_wait` for event retrieval, which additionally provides a timeout mechanism. The difference between *epoll* and Banga's proposal is that *epoll* works on descriptor granularity (i.e. the kernel event structure can be identified via a file descriptor) [88], whereas the solution proposed in [7] works per-process. Hence, *epoll* allows an application to handle multiple distinct event queues (via `epoll_create(int size)`), which is useful for priority-based I/O scheduling.

A weakness of *epoll* is its interface to incrementally update interest sets, *epoll_ctl*, because it only allows to update the event filters for one connection at a time. Hence, updates to all registered connections need to be performed with multiple calls to *epoll_ctl*, which is a system call and therefore expensive. *Kqueue* solves this problem by combining *epoll_wait* and *epoll_ctl* to a single function named *kevent* [80]. This allows to update an array of event filters *and* retrieve a set of signaled events in the *same* kernel call. Moreover, *kqueue* is designed to be a more generic notification mechanism, and therefore allows to monitor nearly any kind of kernel objects (e.g. timers), whereas *epoll* only works for kernel objects that can be identified via file descriptors [80, 88].

A network of *Pony* nodes is not managed in a star topology. The number of nodes to which a runtime process maintains connections to is bounded and relatively small. Hence, the complexity of using event notification mechanisms like above is not justified. Instead, we can meet the performance requirements with simpler mechanisms. Since we only have to manage few connections per node, stateless approaches are sufficient.

3.2.2 Select and Poll

As previously mentioned, the motivation for using complex event notification systems like *epoll* and *kqueue/kevent* is that I/O performance is a problem if many connections need to be maintained. In this case *select* and *poll* are inappropriate. However, as we enforce a network of *Ponies* based on a k-Tree topology, the sockets that each node has to maintain is bounded (one socket for the parent node and $k \in \mathbb{N}$ sockets for its children). In such a case *select* or *poll* are sufficient to satisfy our performance requirements. Maintainability and ease of use as well as reduction of code complexity

becomes more important. Hence, the decision for the *Distribution Actor* implementation had to be made only between *select* and *poll*.

Similar to the notification systems described above, both *select* and *poll* work on the granularity of file descriptors. A file descriptor is a means of identifying the corresponding network connection. Listing 3.1 shows the API of the *select* system call [87], expecting five input parameters. Only the first *nfds* file descriptors are checked. Thus, the value of the first parameter can either be the highest file descriptor plus one or the size of the corresponding file descriptor set (provided with the macro *FD_SETSIZE*) if all sockets in the set should be considered for event multiplexing. At this point it also becomes clear why applications with many connections need to use other mechanisms than select (besides algorithmic complexity). Not only is it a matter of performance but also the fact that the file descriptor sets are of fixed size (on many implementations *FD_SETSIZE* is 1024). Using sets larger than the maximum value results in undefined behavior [87]. In fact, because *fd_sets* are realized as bitmaps, it is not possible to monitor file descriptors with a value larger than *FD_SETSIZE* without additional effort. This limitation can be circumvented by allocating an array of interest sets and pick the corresponding set for the select call based on `FD mod FD_SETSIZE` . However, this would require multiple select calls to cover the entire range of registered file descriptors.

The application registers *interest sets* of type `fd_set` (file descriptor sets) [87] for *read*, *write* and *exception* events. Thus, if a socket should be monitored for read events, it has to be added to the read interest set. Appropriate macros for working with *fd_sets* are provided (initializing, setting and reading). Note that passing file descriptors to the macros must be side-effect free, as the parameter is evaluated multiple times [87]. If no timeout (i.e. `NULL`) is supplied, the *select* call is blocking until one or more file descriptors in the interest set become ready. Effecting a poll that returns immediately is supported by supplying a zero-initialized *timeval* struct. Upon return, the kernel overwrites the interest sets to indicate which of the monitored file descriptors are ready. This also means that for each subsequent call of *select*, the interest sets need to be refilled. It shall be mentioned at this point that a socket monitored for read events also becomes ready on end-of-file (i.e. when a remote node closes the connection) [87].

Listing 3.1: Signal readiness of sockets per interest set: select()

```
1 int select(int nfds,
2            fd_set *readfds,
3            fd_set *writefds,
4            fd_set *exceptfds,
5            struct timeval *timeout);
```

Additionally to the fact that the size of the interest sets is fixed and implementation dependent as well as bitmaps are shared for all sockets, scalability and performance is not optimal as the kernel (as well was the using application) scans the entire bitmaps to check which file descriptors are ready for each call. This is inefficient in a context with many connections, because the set of ready sockets may be relatively sparse in a sense that only a small subset is ready for operation. However, the relative impact of *stateless* mechanisms and their linear complexity is negligible if the set of connections to be monitored is small. Note that we are not required to execute more system calls compared to the protocols described in section 3.2.1.

Although this performance issue still remains for the system call *poll* (at least similarly) [85], the number of sockets that can be monitored is theoretically unbounded. Furthermore, its usage is less verbose. At the same time, the list of registered events does not need to be refilled for each subsequent system call. As code complexity and maintainability was deemed important for the first prototype if performance requirements can be met, we decided to implement the *Distribution Actor* based on *poll*.

Contrary to the *select* API, *poll* allows to pass an array of structures of type *pollfd* rather than maintaining all descriptors using the same bitmap for each event [85]. Each structure contains the file descriptor of the corresponding socket along with a bitmap of registered events. Resetting the information on which events to listen to is not necessary for each call (if the array is properly re-used), because registered events and signaled events are split up into two separate fields - *events*

and *revents*. The timeout has the same effect as in the previous call but only provides a granularity of milliseconds instead of microseconds [85]. However, since we want to effect a poll that returns immediately (even if no sockets are ready), this is not an issue.

Listing 3.2: Retrieve signaled sockets: poll()

```
1 struct pollfd {
2   int    fd;          /* file descriptor */
3   short  events;      /* requested events */
4   short  revents;     /* returned events */
5 };
6
7 int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

In the following, only the mechanism for handling I/O events on sockets in *Pony* is discussed. The "entity" responsible for any node-to-node dispatching as a whole is explained in later sections of this chapter.

Each *Pony* node listens on a re-usable port which can either be passed in as command line argument or can be configured in the operating system's services file (in UNIX based systems this file is located at `/etc/services`). If none of this information is provided, the implementation picks some free port. All sockets maintained by the runtime system are stored in an array of *pollfd* structures. This includes the listener socket at index 0, which is responsible for accepting incoming connections. The size of the array (and therefore the maximum amount of children) can be configured. The handler invoked for connection requests causes a new socket to be created to the requesting remote node. The network topology is based on a k-Tree, which is not visible to any node. Instead, each node only knows about its parent and its children, stored in the same array (i.e. the parent node is stored at index one and so on). The listener socket is configured to be non-blocking, which means that read or write operations will never block, but signal that there is no more data to read or write. Any socket that is newly created for a connection that came in via the listener socket inherits this configuration. The listener socket does not react to any network traffic different from connection requests. If the array storing the sockets is full, new incoming connections are rejected.

Since we are storing every socket within a *pollfd* structure, we can register the corresponding events for which a socket should be monitored. The *readiness* of a socket for write events is implicitly reflected through a message in the *Distribution Actor's* mailbox, as described in section 3.3. Hence, a socket is never monitored for write events, except for connection completion. The non-blocking configuration of the network sockets might cause a call to *connect* to return immediately, if the connection cannot be established without blocking (caused by waiting for the three-way TCP handshake to complete [22]). If so, the socket used for establishing a connection is monitored for writability by setting the *events*-variable of the corresponding *pollfd* structure to `POLLOUT`. After the socket being signaled as writable, we can check whether the connection was established successfully or not (using `getsockopt` [83]).

The listener socket and every other socket created upon successful connection establishment is monitored for read events by setting the corresponding bitmap to `POLLIN`. The states of the sockets can be retrieved using the *poll* system call shown in Listing 3.2, providing the array of *pollfd* structures mentioned earlier. *Poll* returns the number of fired events (or -1 if an error occurred). There is no indication on which of the sockets are ready, so it is necessary to loop over the entire array independently of how many events are indicated. The *revents* bitmap of a non-ready socket is guaranteed to be 0 [85]. Listing 3.3 shows how the corresponding message handler can be invoked for a given event. Note that a connection request causes a socket to be ready for readability.

Listing 3.3: Invoke handler for filter

```
1 int ret = poll(&sock_array, sock_count, 0);
2 pollfd* ev;
3
4 if(ret > 0)
5 {
6     for(int i = 0; i <= sock_count, i++)
7     {
8         if(sock_array[i].revents == 0)
9             continue;
10
11        ev = &sock_array[i];
12
13        if((ev->revents & POLLIN) != 0)
14            handle_read(ev);
15        else if((ev->revents & POLLOUT) != 0)
16            handle_connection_compl(ev);
17    }
18 }
```

The *poll* system call sets the corresponding bit for each signaled event and writes the result to *revents*. Thus, *revents* can be checked using the bit-wise AND operator `&`. Once able to invoke a network handler, it is guaranteed that the corresponding socket operation will not block. It is important to remember that a socket is ready as soon as one TCP packet arrives. Using the *poll* system call, we do not implicitly know how much data is available to read (in the case of *kqueue* this is possible [80]). The consequence is that the read handler is more complex than one might expect, because we do not want to waste CPU cycles desperately trying to read a *Pony* message which might not have arrived in its entirety yet. A solution to this problem is given in section 3.3.1.

A mechanism that allows us to to handle network I/O when necessary is worth nothing if we do not find an appropriate way to decide when we need to actively invoke the described system call. This being a scheduling problem, we need to discuss the challenges of asynchronous I/O multiplexing. An important insight towards a solution is that using any of the system calls mentioned above do not prevent us from checking for I/O unnecessarily, such that the resulting I/O schedule is not optimal. However, in the context of an actor-based language runtime in combination with *Pony's* work stealing heuristics, having no events to handle should be sufficiently rare until the program terminates.

3.2.3 Scheduling I/O Events

In the first design iteration we intended to implement a separate I/O network library for handling network events (reads and writes) from and to remote nodes asynchronously. This API was supposed to include all necessary buffer management, appropriate event handlers as well as a dispatcher to process received messages accordingly. This allows for an autonomous and re-usable component of the runtime system that is highly optimized for networking operations. A single call could then be utilized to check for pending I/O events and to process them, respectively. This approach requires to instantiate an I/O service upon node startup and maintaining a reference to it. A straight-forward implementation would be to let the main scheduler thread maintain the service for network events.

However, this approach makes the problem on *when* and on *which* core to schedule and trigger the handling of such events relatively complex (and possibly inefficient). If only the main thread holds a reference to the I/O service, then we can only handle events if this thread is not busy and before it steals work from other cores. That could cause a relatively high latency between the arrival and processing of a message. Although *Pony* schedules an actor to handle one message per quantum, the I/O service may suffer starvation, because the main thread may theoretically always have actors to execute. A possible way out of this could be to provide a reference to the I/O service for each thread. However, this requires to make the implementation of the asynchronous network library thread-safe, which is unnecessarily complex. At the same time, such a solution can cause

actors that need to send messages to remote nodes to wait due to synchronization mechanisms used for a thread-safe network library (even in the case of a lock-free implementation, because the network layer may be subject to high contention). Especially for a programming language runtime, where executing application actors is of highest priority, this argument makes the implementation of a network library as described undesirable.

The idea is to implement the I/O service as an actor that can receive I/O messages that are supposed to be delivered to remote nodes and dispatches I/O events when sockets become ready. This allows for the I/O service to be scheduled just like any other actor in the system. The fact that the I/O actor may be stolen and scheduled on a different core on each invocation is not an issue, because the probability that the L1 cache is evicted between two invocations should be relatively high, such that we do not degrade performance due to cache misses. Since *Pony* only schedules actors that have messages in their mailbox, the only problem that remains is that the I/O actor may never get scheduled even though sockets are ready to being read (i.e. remote messages have been received) as the event of a message having arrived does not automatically put a message into the I/O actor's mailbox. This can be solved by implicitly sending a "check-for-events" message upon I/O actor creation as well as always sending the same message within the corresponding handler function again. With this approach, the I/O actor will always be scheduled. To avoid busy-waiting, we call a zero-valued *nanosleep* after all necessary I/O work is done.

Having provided a basic solution for I/O scheduling, the responsibilities for the mentioned actor are more than just dispatching of network messages. More importantly, the very same actor is primarily responsible for distributed work stealing. For this reason, we will refer to it as *Distribution Actor*. There is more to network programming than just non-blocking sockets, asynchronous I/O and its scheduling. Efficient mechanisms for determining the type of a message are required. The goal is to avoid that the *Distribution Actor* becomes the bottleneck. The following sections illustrate that achieving this goal is dependent on many different parts of the runtime implementation.

3.2.4 Framed Network Protocol

A common pitfall of programming network applications that TCP is stream-based as opposed to packet-based [22, 69, 71]. Although transferred data is split up into packets, there is no guarantee that a single write operation to a socket corresponds to exactly one TCP packet. This means that a single call to *write* can cause many packets to be sent in which case it might correspond to multiple reads at the receiver side. Furthermore, the standard of TCP as described in [71] allows that many distinct messages could be transferred using the same packet. As a consequence, the receiver of a message has no indication on how much data to read from a socket, not even after the full message has been transmitted, because the network buffer could contain mulitiple messages that came in concurrently. The underlying network stack implementation of TCP only cares about the splitting of messages into packets, and their correct composition at the receiver side (besides some error handling).

This motivates the need for a *framed* network protocol, where each message is preceded with a header that contains appropriate information about the message content in order to determine message boundaries. At the same time, such a protocol allows to efficiently determine the type of a message, which makes the implementation of a message parser easier and more convenient.

Listing 3.4: Pony Message Frames

```
1            * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+- *
2            * | header |     length     | variable length body | *
3            * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+- *
4                1 byte       4 byte            max. 4 GB
```

A *Pony* message frame consists of three fields. The message *header* is one byte in size and is used to efficiently check the type of a message. The *length* field indicates the size of the variable length message body. Both are combined within a structure called `message_frame`. The message protocol needs to support several different types:

- *Application messages* that are sent from a local actor to a remote actor.

- *Runtime control messages* for scheduling, which are required to implement our work stealing heuristics (section 3.5). Additionally, distributed garbage collection and *Pony's* mechanism to connect new peers require their own set of messages.

The goal is to encode the message type within the header and combine it with a flag that allows us to identify a message as being sent from another *Pony* runtime process. Hence, the mechanism that forms the *header* field of a message frame is based on three components:

- `PONY_MASK` is a bit-mask used to check the integrity of a frame header. The mask is not sent across the network and is used to determine the significant bits of the *Pony's* magic flag. Integrity in this context does not mean to match the header against a check sum, but rather to ensure that the incoming message is supported by the receiver.
- `PONY_H` is a magic flag, which is used to identify *Pony* messages.
- The message type is used to determine the purpose of a message. The number of message types to be supported dictates the minimum size of the header field.

These three components are combined to a single bit string and written to every frame's header before sending messages over the network. The actual values chosen for these components depend on each other, because we need to be able to extract every one of them from the frame header on the receiver side.

Assume we decide on a magic flag `PONY_H` with a value of `0x81`, which corresponds to `10000001` in binary representation. We can combine the value of `PONY_H` with the type of the message by using the unset bits of the magic flag, in this case any bit within $[1, 8)$. The value for a message type can now be chosen appropriately. For example, we could decide to define the ID of actor application messages to be `PONY_ACTOR_M = 0x83` and so on.

When sending a message, the frame header can be set to the result of the bit-wise OR between both components, i.e `(PONY_H | PONY_ACTOR_M)`. The value of `PONY_MASK` is used to limit the set of possible input headers as well as to check for the magic flag, independently of the type of a message. An incoming message can be checked to be a *Pony* message using `(header & PONY_MASK )== PONY_H`. If the check fails, the message can be ignored. The same applies for determining the type of a message. Depending on the number of messages to be supported, we could decide to set the mask to `0xe1` (`11100001`, which supports 16 messages).

It is important to distinguish between the mask and the magic flag, because checking `(header & PONY_H)== PONY_H`) succeeds in every case where all bits that are set in `PONY_H` are also set in the header. The result would be a much more fragile network implementation, which could be made to crash easily by injecting corrupt messages. For example, a message header with the value `0xff` would pass the test with using the magic flag only, but not the test using the mask. The application of `PONY_MASK` implicitly rejects message headers that are not supported.

The *length* field in a message frame is used to indicate the number of bytes to be read until the end of a message is reached (i.e. the size of the variable length message body). For the first running prototype, this field is a 4-byte integer, which bounds the maximum size of messages to 4GB. Note that we cannot use a compiler optimization in later version to generate the size of the field, because we also need to support data structures that are of a size which cannot be determined statically (e.g. linked lists). Limiting the size of network messages to some upper bound is fine, because we can schedule actors accordingly if the size of a message exceeds a certain threshold, and therefore it is not a limitation at the language level.

One important aspect of the design of message frames is that we could have also decided to provide more fields that contain information about the type of a message, which would allow to support messages with no message body (and therefore only a single read from a socket). However, in the case of *Pony*, the two discussed fields are the only ones shared between messages of all types, such that we would have ended up with a message frame where most fields are unused for a particular message. This is unnecessary overhead.

Deciding on a particular size of a message frame header or any of the fields involved does not provide a problem for future versions. The implementation is kept within its own component, such

that other parts of the system do not need to be changed when the size of the fields needs to be changed (e.g. for more message types to be supported). Issues with compatibility would only arise if changed the actual shape of the message frame, which is another argument for keeping as little information as possible in the part that precedes the message body.

The communication protocol allows us to dispatch messages between runtime processes on a conceptual level, but there is still some way to go in order to actually achieve efficient message passing between runtime processes. Designing and implementing the distribution part of the runtime is challenging, because it interferes with and depends on many parts of the runtime system whilst being critical for performance. Superficially, some of the solutions seem easier than expected. However, those are the ones that are usually hard to figure out in the first place. Implementing the *Distribution Actor* requires to trade off competing factors against each other. The important part is not only to understand the complexity of the implementation itself, but also that getting to the proposed solutions requires constant improvement and evaluation.

3.3 The Distribution Actor

A *Pony* program that is executed and expected to scale on a computer cluster needs to be started with the command line argument `--ponydistrib`. By doing so, an actor will be created that is in charge of anything necessary for distributed scheduling. This actor is called the *Distribution Actor* and is locally available to each machine in the *Pony* network and is pinned to the host it was created on (i.e. it is never scheduled to another machine). Within this section, we describe the overall structure of the *Distribution Actor*, its external and internal interface as well as some of the data structures used for managing messages and a network of *Ponies*.

We refer to the *external interface* when we talk about the set of message types the actor reacts to and to the *internal interface* when talking about the functions it uses to fulfill its tasks, without exposing them to other parts of the runtime. Although work stealing and the scheduling of actors to other nodes is effectively implemented using this actor, scheduling concerns are described separately in section 3.5. The *Distribution Actor* provides the following external interface:

- `DISTRIBUTION_INIT` is the constructor message used for allocating the necessary heap space and initializing the necessary data structures. This message is implicitly sent to the *Distribution Actor* when it is created upon runtime startup and therefore only sent exactly once per node.

- `DISTRIBUTION_CONNECT` is used to establish a connection to a parent node. A connection is only established if the node is not the *Pony Master* (i.e. not the root node in the tree network). The *master* node must be provided as command line argument in order to initiate the joining process (see section 3.3.2): `--ponyconnect <hostname> <port>`.

- `DISTRIBUTION_PROBE` is used for checking whether there is inbound I/O work to do for any of the sockets that a node maintains to other participating computers. In order to prevent the actor from becoming blocked, the *Distribution Actor* sends this message to itself at the end of each invocation of the message handler for scheduling purposes. No other actor sends this message to the *Distribution Actor*. This message interferes with the termination mechanism discussed earlier (because it is never really "quiet") and will therefore be reconsidered in the remainder of this chapter. Upon receipt of this message, the *Distribution Actor* invokes the *poll* mechanism described in section 3.2.2.

- `DISTRIBUTION_PULL` is sent to the *Distribution Actor* when the local scheduler detects that local cores become available. Following an observation made during development that the free core count may oscillate frequently, in order to avoid flooding the system with messages, this message is sent using an asynchronous timer instead of every time the core count changes.

- `DISTRIBUTION_POST` is used for delegating messages to actors that are located on a remote node. This message is only sent from actor proxies.

- `DISTRIBUTION_CLOSE` terminates the *Distribution Actor*. This causes all remote connections to be closed and all data structures maintained for scheduling to be deallocated. After having received and processed this message, the *Distribution Actor* becomes blocked. Hence, the corresponding runtime process can terminate based on the quiescence scheme described earlier. A *Distribution Actor* is terminated based on the scheme described in section 3.8.

In order to avoid unnecessary code repetition and provide a clear interface, other components (such as actors and the local scheduler) use a set of convenience functions, which encapsulate the sending of the above messages to the *Distribution Actor*.

Various data structures are maintained in order to manage a network of *Pony* runtime processes. This includes two instances of a hash map to keep track of objects received from remote nodes (which is called the *identity map*) as well as a map to store all necessary information for dispatching remote messages to the correct destination actor (called the *proxy map*). A node's ID is stored in a simple integer variable. The advanced work stealing heuristics presented in section 3.5.4 require that each node maintains the free core count of all its descendants (where *descendant* is the transitive closure of the children relation). As previously mentioned, directly connected nodes are maintained in an array of *pollfd* structures, which at the same time is used to poll for asynchronous I/O events. The first element of this array is the listener socket for incoming connections.

The *internal interface* of the *Distribution Actor* (of which parts are implemented in `network.c`) consists of the necessary event handlers for asynchronous I/O as well as functions to determine the next hop for a network message (if necessary). The read handler for network I/O (`handle_read`) is invoked for every socket that was signaled for read events. The write handler (`handle_write`) is only invoked to check for connection completion but not for `DISTRIBUTION_POST` or `DISTRIBUTION_PULL` messages, because the *Distribution Actor* attempts to write any data resulting from these two messages to the corresponding network socket without giving up (i.e. during the quantum at which either of the two control messages is dispatched). If the listener socket is signaled for readability, then the handler responsible is `handle_accept`, which is used to accept new connections or to trigger delegation of a joining slave node (section 3.3.2). Furthermore, a handler function for each type of *Pony* runtime control message is provided. Whenever a remote message is received, the type of the message is determined based on the frame header discussed earlier and the corresponding function is picked to process the message.

Whenever the *Distribution Actor* decides to migrate actors to another node (or in fact to another part of the network), it attempts to steal as many actors as necessary from the local scheduling queues. Note that we might be not be able to steal any actors. The *Distribution Actor* has no possibility to determine whether an attempt to steal work is going to be successful or not, because we cannot efficiently inspect all available scheduling queues. However, the CPU time invested in attempting to steal work is entirely wasted, because we have updated the free core count of at least one child node, which is important for termination. We will come back to the relationship between free core counts, actor migration and termination in section 3.5.4 and 3.8

One important fundament for the performance of *Distributed Pony* is the underlying networking implementation. In the context of a programming language runtime, there might always be work to do. Wasting CPU time through waiting for I/O is therefore undesirable. *Pony's* non-blocking network API – discussed in the following – is designed to reduce the overhead involved with networking I/O and ensures that the amount of time spent executing system calls for network socket operations is kept to a minimum.

3.3.1 Non-blocking Sockets

We have previously mentioned that one requirement for *Pony's* network I/O capability is that we should spend as few cycles as possible reading or writing to the network. An important part of this is to use non-blocking sockets. This guarantees that no socket operation will ever block, independently of the amount of data we attempt to read from it. For example, upon every network message we first need to read the part that precedes the message body, which is five bytes in size. Theoretically, a system call to any socket reading the frame header could return anywhere within the five byte boundary. Note that it cannot return having read nothing from the socket, because

in this case the socket would not have been signaled that it was ready (due to the *poll* mechanism in use), except when a remote node closes the connection. The same applies to write operations, because the outgoing buffer of a socket may be full. At the same time, we need to minimize the amount of system calls, because switching to kernel mode is expensive. As a result, the read and write handlers of the networking API (`network.c`) become considerably more complex. We need to be able to give up reading and continue at that point when more data is available, for all sockets maintained by a node at the same time. This allows reading as much as we can get from all sockets whenever the *Distribution Actor* is scheduled, and give up the quantum once all ready sockets haven been visited. Of course, this also means that we cannot read all of the data that might be available in total, because a socket may have become ready again after we moved on to the next network connection. From a scheduling perspective this is fine, because we do not want to extend the quantum of the *Distribution Actor* indefinitely. Hence, the maximum time spent on reading from a socket is equivalent to the time it takes to read an entire inbound buffer from each socket. The size of the inbound buffers is fixed and can be configured. The tree network topology bounds the number of connections each node has to maintain and therefore – at least to some extent – the worst-case time complexity of the networking handlers can be influenced. Note that this property only holds if a socket's inbound buffer is capable of holding the largest message of a given application. For this reason, it can be worth tuning an application by changing the corresponding buffer sizes. However, in most cases, the default settings (which are implementation dependent) should be sufficient.

For each socket, we maintain a separate inbound buffer (Listing 3.5). Once a socket becomes ready for reading, we keep track of the number of bytes to be read for a specific part of a message as well as an offset pointer to the location within the buffer that we are writing to for a given message. Depending on the value of `collect_started`, this pointer either points within a structure of type `message_frame` or within the allocated buffer for the message body (`msg_body`). Remember that reading a message requires at least two read operations, because we first need to acquire the frame header (where the size is statically known), before reading the message body. For convenient use of pointer arithmetic, the offset pointer is chosen to be of type `char*`.

Listing 3.5: Network Message Buffer

```
1 typedef struct message_buf
2 {
3     size_t bytes_left;
4     char* offset;
5     bool collect_started;
6     bool header_complete;
7     message_frame* frame;
8     void* msg_body;
9     remoting_fn dispatch;
10 } message_buf;
```

The corresponding read-handler function (`collect(...)` of `network.c`) is recursive and reads as much data as possible from each ready socket. Pseudo-code is provided in Appendix A. If a given message buffer is not indicated to be involved in a current collection (`collect_started`), we know that the next bytes to read need to be a frame header.

After the frame header has been read successfully, we check for the magic flag of the message and decide whether to ignore the message or not. If accepted, the length of the message body can be determined using the length field. The function gives up when the read call blocks and returns zero, which causes the *Distribution Actor* to continue with the next ready socket (or to give up its quantum if no more socket needs to be visited). Once a message has been read completely, the collection function sets the dispatch handler `remoting_fn` to be invoked. This handler depends on the type of the message (scheduling message, application message etc). The return value is a pointer to the receiving actor. How the receiving actor is determined varies depending on the type of the message, and is discussed in section 3.4 and 3.5.

Minimizing the number of write system calls for sending messages to remote nodes follows a similar approach, but is largely dependent on the implemented *stream buffer* used for serialization.

We postpone the discussion of the write-handler function to the next section. The difference between reading and writing is that we do not give up writing when the call blocks. Instead, we keep running in a loop until the entire message was written to the target socket's output buffer. The right setting of the output buffer sizes determines whether we have to attempt writing multiple times or not. Otherwise, we would be required to maintain a separate buffer to store outgoing messages until they have been successfully flushed.

Depending on the amount of actors running on a node, it might take some time until the distribution actor is scheduled again, which would degrade the throughput of the distributed application unnecessarily. The occurrence of a situation where we need to retry writing to a socket should be rare, because the operating system has a time window to flush a socket's output buffer until the *Distribution Actor* becomes scheduled again. This window can essentially be influenced by the scheduler if using a timer to probe for network I/O (instead of a message).

3.3.2 Connecting Slave Nodes and Routing

Before scheduling actors on a set of distributed machines, we need to be able to join nodes to an existing network of *Ponies* as well as a scheme to route messages from source to destination. The mechanism to join new nodes is especially important, because it has a large impact on the amount of configuration and administration required to set up a *Pony* cluster. Similar to the philosophy we follow for developing the language specification (abstracting from topological and technical details), we want a *Pony* cluster to be easy to configure and to maintain. Setting up a distributed network of *Ponies* is possible without detailed knowledge about the underlying computing resources. The runtime system imposes as few constraints as possible on the underlying hardware infrastructure.

There are many different approaches to solve these problems. In fact, the first discussions about the desired feature set for *Distributed Pony* did not consider these concerns in great detail. Why this was the case becomes clear when discussing the problem of work stealing actors in a distributed context (section 3.5). Only after having fully understood what it takes to implement a distributed scheduler for actors, it became clear that nearly none of the problems that came up can be solved in isolation. For example, as discussed in the remainder of this chapter, it is not self-evident that (besides configuration convenience) the joining mechanism and routing of messages is inseparable from object identity comparison, garbage collection and causality.

We will reflect on possible alternatives when discussing the distributed scheduler implementation and point out why they *had* to be ruled out. Also, the first prototype does not take any node-to-node latency into account. In order to improve data locality, future versions of *Pony* might use an adapted scheme, such that a parent is picked based on the network delay or some comparable measurement ("Who can ping the joining node fastest?"). In the following, we will discuss a solution on how slave nodes can be connected to a cluster of *Ponies* (which shall suffice for the purpose of this project) as well as how messages are routed within the system.

Every cluster of *Ponies* requires one of the runtime processes to be determined as *master* node, which is the root within the tree network topology. The runtime system should be able to construct the tree network automatically. Therefore, every new slave node connects to the *master* and is either accepted as a direct child, or delegated to some node further down the tree. Once a node with a free slot for a child node is found, the joining node chooses this node as its parent and the initial connection that was established to the *master* for joining the network is closed. Hence, provided that the tree is balanced, the joining algorithm of *Distributed Pony* has a worst-case time and space complexity of $O(\log n)$, where n is the number of currently participating nodes.

The difficult part is to decide at each node via which path a joining runtime process should be delegated to a parent node. This is important, because we want that the tree of runtime processes is *almost* balanced. We refer to it as *almost* balanced, because each time a new level is added to the tree hierarchy, the tree becomes balanced once all nodes have a fully utilized child-set. This property is important for the performance of the work stealing heuristics, which are discussed in section 3.5. The *master* node assigns an ID to each new participant by simply incrementing a counter. The ID of the root node is 0. Once the maximum number of child nodes for the root is reached, delegation is decided using a scheme that allows us to fill up the tree in a breadth-first

fashion, whilst guaranteeing our desired *almost balanced* property. Thus, we need to develop a recursive algorithm that, given a node ID n, returns the path to that node ID, in a tree where each node has up to k children.

First, we develop a formula that computes the number of non-root nodes in a k-Tree of depth d:

$$All(d) = \sum_{i=1, i \in \mathbb{N}}^{d} k^i \tag{3.1}$$

For delegating a node from the root to an appropriate parent node, we need to be able to derive the path from the root to a nodes location within the tree from its ordinal number. First, consider the following formula, which computes the ordinal number of a node based on the path via which it is reachable from the root node:

$$\begin{aligned} &Ord : \mathbb{N}^* \rightarrow \mathbb{N} \\ &Ord(a_1...a_d) = All(d-1) + (((a_1 * k + a_2) * k + a_3)) * k + ... + a_{d-1}) * k + a_d + 1 \end{aligned} \tag{3.2}$$

For example in a tree with $k = 2$, the ordinal number of a node located at path `0.0` is $Ord(0.0) = 2 + 1 = 3$. Similarly, the ordinal number at path `0.1` is $Ord(0.1) = 2 + 1 + 1 = 4$. For joining a new node to a *Pony* cluster, we want to compute the path to a new node's parent based on the new slave nodes ordinal number, i.e. the inverse of the above formula:

$$\begin{aligned} &Ord^{-1} : \mathbb{N}^* \rightarrow \mathbb{N} \\ &\forall m \in \mathbb{N}^+ : Ord^{-1}(m) = a_1...a_n \text{ such that } \forall i \in \mathbb{N}^+ : 0 < a_i \leq k \wedge Ord(a_1...a_n) = m \end{aligned} \tag{3.3}$$

We need to develop an algorithm that computes Ord^{-1}. Hence, the objective is to find a number p such that $All(p-1) < m \leq All(p)$, and compute:

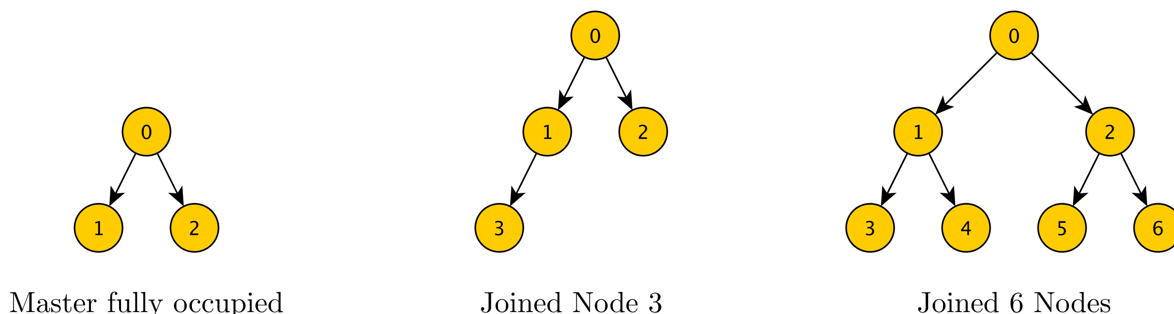
$$\begin{aligned} &b_p = m - All(p-1) - 1 \\ &\text{for i in range(p, 1):} \\ &\quad a_i := b_i \bmod k \\ &\quad b_{i-1} := \frac{(b_i - a_i)}{k} \end{aligned} \tag{3.4}$$

Given a binary tree, following this algorithm, a new slave node with an ordinal number (node ID) of 5 would be delegated to a parent node via the path $a_1.a_2 =$ `1.0`, i.e node 5 becomes a child of a node with ID 2, as shown in Figure 3.2. Note that the path for joining a new node can be computed efficiently if k is a power of 2, because then the *modulo* operation and *division* can be implemented using a bit mask in combination with bit shift operators. For this reason, it might be sensible to require from the programmer that k *needs* to be a power of 2.

The actual assignment to a parent for a node joining a *Pony* cluster depends on the number of children a node is allowed to maintain, and can be dynamically configured for the *master* node using the command line argument `--childcount <n>` (which is increased until n is a power of 2). Slave nodes inherit this configuration. Not only does this approach allow to easily set up a *Distributed Pony* cluster, but also provides the basis for a routing scheme. Figure 3.2 illustrates this process for a binary tree.

The *master* node accepts new connections until the maximum number of children is reached (in the case of a binary tree $k = 2$). A new slave node is told its ID independently of whether the *master* accepts the connection or delegates the slave further down the tree. If the node that the new runtime process attempts to pick as parent is fully occupied, it sends a delegate message to the requester, containing all necessary host information about the next possible parent. For example in Figure 3.2, the fully occupied *master* node – knowing that the new peer must be of ID 3 – would reject the connection request, and delegate the new node to its child with ID 1. Node 1 has no children yet and therefore accepts the connection. Following the same approach, the next slave node with ID 4 would also become a child of 1. Figure 3.2 shows the result after six nodes

Figure 3.2: Joining new Slave Nodes



have joined the network. Evidently, the next node to accept a new connection would be node 3. Note that there will always be some tuning necessary by the *Pony* programmer setting up the cluster, because depth (i.e. latency) might be more harmful than breadth (i.e message forwarding per node), in which case the number of maximum children per node should be increased.

The same delegation scheme can be used for routing. Every node can compute the path to a node from the root. If a sending or intermediate node finds itself on that path, it delegates the message towards a path down the tree. Otherwise, a message is delegated to a parent node.

A limitation of this approach is the tree structure itself, because it fails in utilizing all (possibly) available network channels. The network throughput is lower than what might be technically available. For example, a message from node 0 to node 5 would be routed using the path in the tree, instead of a channel that might directly exist between the two nodes. More importantly, the worst-case time and space complexity for routing messages is in $O(n)$, because routing from node 6 to 3 requires n steps and messages, even if there were a direct channel between the two. Although there is a tree labeling algorithm to cushion this effect [119], we will stick to the labeling scheme described above. This may be a tough call, but we are interested in a tree network topology for a specific purpose, as described in chapter 4. Furthermore, the proposed algorithm may cause paths to be used in an imbalanced manner (due to its breadth-first nature). We might change the algorithm in future version of *Distributed Pony*, such that tree is filled up in "gaps" (i.e. node 4 would become a child of 2 and so on).

Future work will require to determine whether this type of network topology is a major limitation for *Pony's* distributed computing performance. However, we are convinced that it is possible to make up for the flaw described above with fine-tuned scheduling heuristics. An outlook on how these may look like is provided in chapter 6. How *partial failure* can be handled using this scheme without reconstructing the entire labeling is discussed in section 6.2.

A slave node, after having joined a *Pony* cluster, should receive work to process as soon as possible. However, before getting to that, it is required to provide serialization and deserialization of any structure that might be sent over the network.

3.4 Serialization and Deserialization

Serialization and deserialization of actors, actor references, arbitrary objects and primitive types is an important part of the runtime implementation, because it considerably influences the overhead introduced by distributing an application. Having this in mind, we were convinced that it is important to avoid that the *Pony* serializer becomes the bottleneck. It seemed obvious that it should be possible to serialize as well as deserialize actors and messages in parallel. Whereas in many cases this approach would simplify various aspects of deserialization (such as allocating the necessary heap space for actors), there are several problems, which are illuminated in the remainder of this chapter.

Also, because we develop the runtime system prior to a compiler and a complete language specification, the implementation described is more complex than necessary, because we have no

type system support to automatically generate the necessary functions. For this reason, we have developed a recursive *trace* mechanism that scans a given structure and executes the necessary write operations, as described in section 3.4.2. Another aspect is that we cannot efficiently determine the size of a data structure prior to serialization. Hence, a buffer type that can be written to while scanning a data structure is required, without the need of pre-allocating memory space for the entire structure. This class of buffer types is also referred to as *stream buffers*.

3.4.1 Stream Buffer and I/O Vector

Not only is the buffer implementation vital for efficient serialization, but also for writing the data to a network socket. The fact that we do not know the size of a structure a priori (and therefore cannot pre-allocate memory), causes that the buffer might be required to chain-up multiple chunks of data. Additionally, we cannot align the chunks of the stream buffer in advance, which can result in a structure (e.g. a primitive type) to be written across chunk boundaries (i.e. the value has been partially written to one chunk and ends in another chunk). However, for efficiency reasons, we do not want to be required to care about any type alignment, because we want to have stream chunks of static size, such that we can make use of *Pony's* pool allocator.

As system calls are expensive, it would be undesirable to execute multiple calls to the socket-write function. Preferably, we want to invoke the *write* system call *exactly* once, independently from the size and number of chunks of the stream buffer. We cannot absolutely guarantee that the write call only needs to be called once, because the number of bytes that can be written without blocking also depends on the filling level of a socket's output buffer. However, we can ensure that the write is performed with as *few* write-calls as possible. Moreover, by appropriate scheduling *and* tuning of the socket-buffer sizes, requiring multiple calls should be the exceptional case.

Pony's stream buffer (provided in `streambuf.c`) is initialized for a specific type of message and is therefore given a one byte message header. This allows to precede the buffer with an appropriate `message_frame` before writing to a socket. By default, the stream buffer implementation allocates chunks of 1KB in size (one at a time). In order to be able to stream values, it is necessary to maintain a pointer that moves forward as we go along and serialize a structure. Once a chunk is fully utilized, a new chunk is allocated and appended to a singly-linked list. Each chunk is contained within a structure of type `iovec` (I/O Vector), containing a base pointer and a length field, as shown in Listing 3.6. This structure is provided by the operating system for *scatter-gather* I/O, which is a mechanism to write multiple buffers to a socket (or file descriptor) that are separated in memory using the system call `writev` [89].

Listing 3.6: Structure iovec as defined in sys/uio.h

```
1 struct iovec {
2     void  *iov_base;     /* Base Pointer */
3     size_t iov_len;      /* Number of bytes */
4 };
```

The base pointer points to one chunk of data. The length field indicates the number of bytes that have been written to the corresponding chunk (and has therefore a maximum value of 1024). It is then possible to materialize the stream buffer to an array of `iovec` elements, and write the entire array to a socket with a single call to `writev`, instead of being required to trigger a system call for each chunk. This strategy obviously pays off for structures that span across multiple chunks, as the overhead associated with system calls can be considerably reduced.

Note that materializing a linked-list of `iovec` structures does not require to additionally allocate heap space. Since we know how many chunks (and therefore how many `iovecs`) have been used, we can simply copy the vector to a variable-length array on the stack of the corresponding function that writes the buffer to some socket. Once the vector is written successfully, the buffer structure can be destroyed. Memory management is contained within the stream buffer implementation and only uses the pool allocator discussed earlier.

The buffer itself does not need to be *thread-safe* if parallel serialization is desired. We could simply maintain a thread-local variable that holds a pointer to a buffer structure, such that every

thread would manipulate a different *stream buffer*. The C-runtime guarantees that a variable annotated with `__thread` is allocated per extant thread [46]. Note that this type specifier can only be used alone, or with the storage classes **static** or **extern** [46].

The first running prototype of *Distributed Pony* – which did not support object identity comparison nor garbage collection – supported parallel serialization *and* deserialization of actors and messages in order to keep the time spent executing the *Distribution Actor* as short as possible. Each actor was responsible to serialize outgoing and deserialize incoming messages on its own. We will come back to this issue in section 3.5 and 3.6 and explain why we cannot support parallelism outside the context of *Distribution Actors*.

We do not consider this as a major limitation to the performance of *Distributed Pony*, because we could imagine a more complex scheme of multiplexing I/O events, where each node schedules multiple *Distribution Actors* instead of just one. By doing so, we would be able to serialize and deserialize in parallel safely. However, this requires major implementation efforts on the scheduler and on a synchronization scheme between *Distribution Actors*. Before embracing these challenges, we should attempt to master the complexity of transparent distributed scheduling whilst maintaining desired language properties (such as causality). Hence, a more complex I/O scheme is subject to future work.

Deserialization does not require to make use of a *stream buffer*, because we can allocate space on the *Distribution Actor's* heap. The *length* field of a message frame is always an overestimation, because every network message contains management information. Instead, memory is allocated as we go along. The scanning of a structure to be serialized or deserialized is implemented by *Pony's* trace mechanism.

3.4.2 Pony Trace

Implementing a recursive trace mechanism for serialization and deserialization is necessary, because we have not implemented a compiler yet. Thus, type system support is not available, which would allow us to generate the required procedures. At the same time, this mechanism can be used to update the reference counts for garbage collection when sending and receiving messages. Therefore, *Pony Trace* supports several *trace modes*, as shown in the definition of `trace_t` in Listing 3.7.

Listing 3.7: Pony Trace Modes

```
1 typedef enum
2 {
3   TRACE_MARK,
4   TRACE_SEND,
5   TRACE_RECEIVE,
6   TRACE_SERIALIZE,
7   TRACE_DESERIALIZE
8 } trace_t;
```

`TRACE_MARK`, `TRACE_SEND` and `TRACE_RECEIVE` are used for reference counting purposes and the remaining two for serialization and deserialization. The behavior of the trace implementation (`trace.c`) can be influenced by setting the mode accordingly. A full trace of a structure can be started by calling `pony_trace`, as shown in Listing 3.8.

Listing 3.8: Start a Pony Trace

```
1 void pony_trace(void* p, trace_fn f, pony_mode_t mode);
```

For serialisation, `p` points to the location to read from and for deserialization to the location to write to, respectively. The argument `f` is a pointer to the (top-level) trace function that corresponds to the type indicated by `pony_mode_t` (implemented as *enum*). The mode is important for determining how a structure has to be handled (primitive type, actor or complex type). The following example illustrates the use of `pony_trace` for a message with primitive arguments as well as for an actor structure.

Remember that an application message is described by the number of arguments, a set of trace functions as well as the corresponding type mode per argument. A message with a 64-bit integer as argument can therefore be defined as shown in Listing 3.9. Assuming that `arg` is a pointer to the argument of the corresponding message, we can serialize it by invoking `pony_trace(arg, m_64prim.trace[0], m_64prim.mode[0])`. Of course, in the case of deserialization, `arg` would need to point to some destination to which we want to write the message argument. Serialization however, implicitly knows the target to write to, using the stream buffer discussed earlier.

Listing 3.9: Message with 64-bit integer argument

```
1 static message_type_t m_64prim = {1, {pony_trace64}, {PONY_PRIMITIVE64}};
```

Primitive types are easy to handle and no recursive calls to any subsequent trace functions are necessary. An actor structure however can be of any shape, containing multiple data members (which themselves are eventually primitive), pointers to complex types or references to other actors. Similar to message types, the type (`actor_type_t`) of an actor is also given a pointer to a trace function. This function is the top-level trace entry point, which is given to a call to `pony_trace` upon serialization or deserialization. An example for such a function and an actor holding a primitive type as well as a reference to another actor is shown in Listing 3.10.

Listing 3.10: Top Level Trace Function for Actor

```
1  typedef struct some_actor
2  {
3      uint64_t prim_type;
4      actor_t* actor_ref;
5  } some_actor;
6
7  void trace(void* p)
8  {
9      actor_type_desc* d = p;
10     pony_trace64(&d->prim_type);
11
12     /* trace function is NULL because references
13      * to actors are never materialized */
14     pony_trace(&d->actor_ref, NULL, PONY_ACTOR);
15 }
```

A full trace of the actor structure above can be triggered using `pony_trace` providing the corresponding top-level function as argument, which itself invokes a trace function for each data member. Note that `pony_trace64` is a convenience function, such that it is not necessary to switch-case on the mode of a primitive type. In order to stick to the interface of *Pony's* trace module, we should invoke `pony_trace(&d->prim_type, pony_trace_primitive, PONY_PRIMITIVE64)`. An actor could also hold a reference to objects of any kind. Each complex type requires to define its own trace function which can be given to `pony_trace`. Actor references are never provided a trace function, because we need to forbid that pointers to actors are flattened. Since actors are "entities" of execution, they contain a part of the state of the overall application and therefore we cannot just copy it to some other machine. Actors are only serialized when they are subject for being scheduled remotely.

References to actors instead are more difficult to handle, because we need to ensure that remote actors, which may receive an actor reference in a message, can use this pointer to send messages to the referenced actor. The same applies to an actor which becomes scheduled to another machine. This is a non-trivial problem, because we need to be able to deliver any messages which are sent using references contained within remote messages (or actor data members) to the same actor located on the machine where the received reference is valid.

A similar challenge comes up for handling object references. The problem in this case is not around serialization or deserialization, but comparing the identity of two objects. Objects are identical if their location in memory is identical. However, once an object is serialized and sent over the network, it technically becomes a new identity on the remote machine after deserialization.

Note that this problem is particularly difficult, because a remote actor might receive the same object within a message from multiple actors which themselves are scheduled on different machines. Moreover, an actor might have received the identical object in a previous message, so overwriting the previously received object with an object of the same identity in subsequent messages would not be correct. For efficiency reasons, we cannot flood the system with messages in order to find the original object of a remote copy. Preferably, we do not want to send any messages at all, such that identity comparison does not become considerably slower in a distributed context. We will come back to this issue in section 3.6.

The procedure for serializing or deserializing arbitrary complex types is equivalent to the one described for actors (in fact, an actor is essentially a complex type). The actual data that is written and read while tracing any (complex) type is not just the data contained within the structure, but also management information required by the runtime system. This information is important for the proposed scheduling implementation as well as for object identity comparison. Therefore, we defer the discussion of their semantics to the corresponding sections in the remainder of this chapter.

3.5 Distributed Work Stealing

The discussion on implementing a work stealing scheduler for *Concurrent Pony* (section 2.6.3) demonstrated that implementing a powerful scheme to distribute the workload on a single machine is hard. This is even more the case in the context of *Distributed Pony*, because messaging over a (comparatively slow) network is involved. The design of a distributed scheduler should keep in mind that the performance of the entire system also depends largely on how the underlying networking resources are managed. In fact, it is not guaranteed that applications will run faster, just because more resources are available. Scheduling parts of an application on different machines can make an application slower compared to a single-host environment if most time is wasted for network I/O. On account for this, the scheduler for *Distributed Pony* was developed with certain characteristics in mind:

- **Transparent migration of actors**: No actor should be required to maintain any topological information. If referenced actors are migrated, the behavior of the referencing actor(s) should not change. Thus, it should be possible to treat migrated actors just like local actors whilst maintaining the consistency of the application.
- If a message is sent to a remote node via multiple intermediate hops, it should not be required to serialize and deserialize more than once.
- **Leverage data locality**: Nodes should first attempt to steal work from machines that are closest in the network (in terms of latency). Note that this depends on an optimized version of *Pony's* join mechanism. As for now, we assume nodes are closest if they are a child or parent to the node attempting to steal work.
- **Distribute workload** in a **balanced** manner. This is the trickiest part, because it may be required to migrate actors to a node which may not have attempted to steal work. We need to provide a scheme to find nodes in the tree that are less occupied. The scheduler should be optimized for **throughput** (which in the long run also optimizes for **latency**).
- **Maintain causality**: We have previously mentioned that we attempt to guarantee causality within *Distributed Pony* by construction and with no software overhead. During development of *Pony's* distributed scheduler, it became evident that transparent actor migration has to be designed carefully. We will come back to this point in section 3.5.1 and why causality must be considered for migration.

First, we discuss very basic distributed work stealing heuristics, which we have implemented for a first running showcase in early stages of this project. Many of the implementation efforts carried out for the scheduler largely benefit from guaranteed causality in the concurrent version of *Pony*.

Hence, the following sections should also be understood as an example to show how important and valuable causality for developing concurrent and distributed software systems can be.

The main challenges are not detecting when to steal work from remote nodes or how to pick a possible victim, but complexity arises when thinking of a scheduling scheme that allows to *uniformly* distribute actors in a network, such that the throughput of the system can be maximized. Also, handling references to actors, which can be contained within a message or be a data member of an actor that is subject for migration, is a non-trivial task. The following sections attempt to incrementally discuss the proposed distributed work stealing scheduler, because this seems the only way to understand that implementing an efficient distributed runtime requires to tune little details and understand the consequences of different approaches in order to pick the *least-worst* compromise (if necessary).

3.5.1 Migration and Actor Proxies

In order to extend a programming language runtime based on actors for (transparent) distributed computing, it is necessary to be able to migrate actors from one machine to another. Figure 3.3 shows a simple example of a network including three nodes, each of which have the same core count. Assume that the *master* node invokes a test application which forks as many actors as there are cores available in the entire network. Evidently, the scheduler should migrate four actors each to node 1 and 2 in order to achieve an optimal workload distribution.

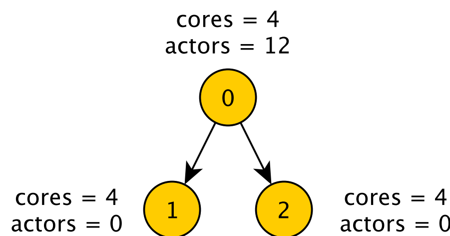


Figure 3.3: Pony Network - Three Nodes, 12 cores and 12 actors

A basic approach could be for each node to register its parent node as possible victim. Once an idle core is detected, a node can simply send a work stealing request message to its victim node, containing the free core count in the message body. Upon receipt of such a message, a node attempts to steal as many actors from the local scheduling queues as requested (if possible), serialize the actor(s) and send the corresponding data to the requesting node.

Since there are possibly other actors referencing one of the actors to be migrated, we need to ensure that these references stay valid *and* that messages are delegated to the corresponding remote site once migration is completed. The key idea is that an actor *becomes* a *proxy* once it has been picked for migration. This is reflected through circumventing the actor's message dispatch handler (by simply setting a boolean flag). If an *actor proxy* is scheduled, a delegation primitive is called which hands over the message to the *Distribution Actor* discussed earlier. For this to work, the *Distribution Actor* needs to maintain information to which node an actor was migrated to. Additionally, a receiving remote node must be able to determine the corresponding target actor. The memory address the migrated actor had on the origin node is not sufficient, because it may be ambiguous between multiple nodes. Moreover, upon receipt of an actor structure, a remote node needs to know the type of the actor received, such that the corresponding data structure can be initialized. For this reason, we introduce *Pony global IDs* in order to determine the *identity* of an actor.

Remember that each node joining a *Pony* cluster was given a node identifier. We can uniquely identify an actor by combining the node ID with the memory address the actor has on the corresponding node. This information can be encoded in a single 64-bit integer. The memory subsystem actually only uses 48-bits for main-memory addressing. Also, *Pony's* pool allocator aligns any al-

location on a 64B boundary. Hence, the lower 5 bits of any memory address are always zero. This allows us to store the memory address of an actor in the lower 43-bits of a 64-bit integer. The remaining 21 upper bits are reserved for the node ID. Note that this combination limits a *Pony* Cluster to manage a maximum number of about two million peers, where each peer may have an arbitrary number of cores. The *Distribution Actor* simply maintains a hash map (provided in `hash.c`) that maps the local memory address of an actor proxy to the corresponding global ID (which contains the destination node). To find the correct target actor for a message, a receiving node maintains the reverse mapping. The encoding is shown in Listing 3.11, where `memory_addr` is the address of some structure (e.g. an actor) and `node` the ID a node had received from the *master* when joining the network. The node ID is sufficient to determine the next hop a message needs to be delegated to.

Listing 3.11: Encoding of Pony Global IDs

```
1 /* bit-wise operators:                                    *
2  * | logical or; & logical and; >>, << right and left bit shift */
3
4 uint64_t glob_id = (memory_addr >> 5) | (node << 43);
5
6 uint64_t memory_addr = glob_id << 5 & ((1 << 48)-1);
7 uint32_t node = glob_id >> 43;
```

Each byte stream of a serialized actor is preceded with the global ID mentioned above and a type ID. The type ID is used to retrieve the corresponding `actor_type_t` structure, which was used for `pony_create`.

The runtime system needs to guarantee that delegated messages arrive at the remote side *after* migration is completed. The *causality* property of *Concurrent Pony* together with the TCP network protocol in use enable us not to actively care about this problem. An actor becomes a proxy by setting a boolean flag. During serialization, an actor is removed from the scheduling queues. Therefore, there cannot be any concurrent state changes while serializing the actor. The *Distribution Actor* writes the serialized actor structure to a network socket *before* putting the actor (which is now a proxy) back on one of the scheduling queues. Hence, any delegate messages will arrive in the *Distribution Actor's* mailbox after the actor has been sent to some remote node. As a result, the byte stream of the serialized actor is sent to the target node *before* any of the messages delegated via the proxy. TCP guarantees that messages sent over the *same* channel arrive in order at the remote site [22, 69]. Hence, migration is completed before any messages need to be dispatched, because the remote node first handles the migration message and then anything else.

If all 12 actors shown in the example above are not blocked upon the receipt of the two work stealing requests, the described basic scheme works perfectly fine, and results in an optimal workload distribution. Figure 3.4 shows a more realistic example, which we use to discuss the limitations of the simple approach described above. A network of *Ponies* does not need to be homogeneous in the sense that the number of available processor cores can vary between participating nodes. Moreover, a *Pony* network (most likely) consists of multiple levels in the tree hierarchy instead of just one. The basic scheme does not cater for any transitive actor migration. In the example provided, node 1 will attempt to steal four actors from the *master*. Node 3 however will never receive work from 1. This is obviously problematic, because we would have unused resources. Therefore, a scheme where each node attempts to steal work just for its own is not sufficient.
Most importantly, although the *master* node starts the application and slave nodes have no implicit main actor, migrated actors may create new actors. Consequently, it is possible that for example node 2 might have many more actors to process than available cores. If the *master* node is always busy, node 2 would remain overloaded, independent of how many cores may be idle in the entire network. The scheduler should be able to detect such imbalances and redistribute actors appropriately. Hence, migration should both be implicit, where some machine requests work, and explicit, where a machine is actively given work without having asked (i.e. some form of *push* and *pull* scheme).

However, there is a problem with this scheme that came up during development. Although we

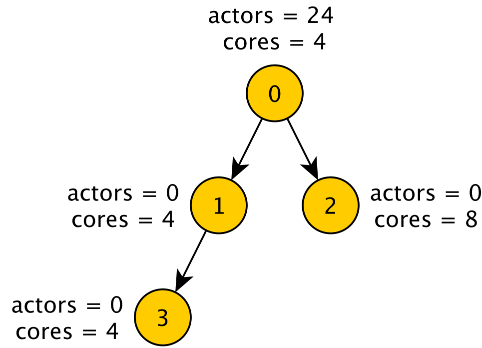


Figure 3.4: Pony Network - Deeper Tree Structure

can guarantee causal message delivery in a distributed setting (see chapter 4), migration can break causality. In the following, we provide an example scenario were this is the case and based on that develop an extended migration protocol that maintains the desired causality property.

3.5.2 Causality-aware Migration Protocol

Figures 3.5 and 3.6 illustrate an example scenario where actor migration would break the causality property of *Pony*. Consider two actors `A1`, `B1` located on `Node A` and a third actor `A2` located on `Node B`. `A1` sends a message m_1 to `A2` via the local proxy `A2p`. After that, the same actor sends a message m_2 to `B`. In response to message m_2, actor `B` sends a message m_3 to `A2` via the same local proxy `A2p`. Causal message delivery requires that we guarantee that message m_1 is enqueued in the mailbox of `A2` before m_3. In the scenario shown in Figure 3.5 causal delivery is guaranteed, because delivery at Node `A` is causal and messages arrive in order at the *Distribution Actor* `Db`. In fact, causality remains guaranteed even if `A2` is migrated to another node where a proxy for `A2` is not present.

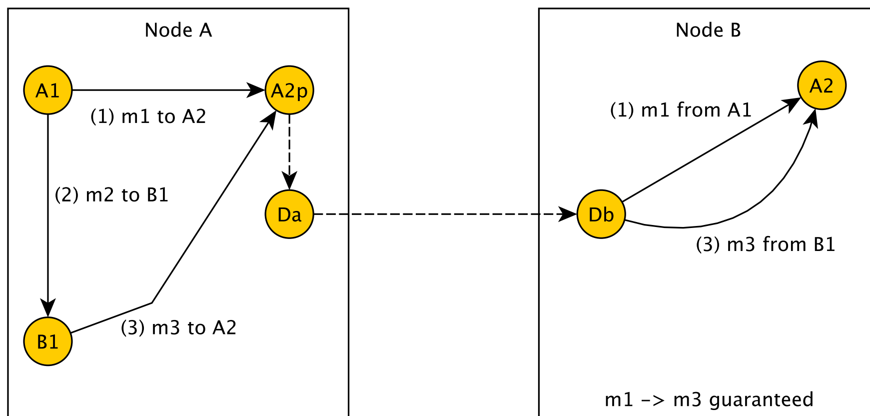


Figure 3.5: Causal Message Delivery - Guaranteed

However, there is a special case if an actor is supposed to be migrated to a node where a proxy for the same actor already exists. Consider the same messages being sent as in the previous example. Before `A2` is able to process m_1, `Db` might decide to migrate `A2` to Node `A`, which already contains a proxy `A2p` for the same actor. Note that the migration process does not serialize the message queue of an actor. Instead, messages are eventually delegated to a migrated actor's new location. Once `A2` arrives at `Node A`, the proxy becomes a local true actor. The problem is that there is no guarantee that m_1 is delegated to the new location of `A2`, before m_3 of `B` is enqueued in the message queue of `A2`, which is now not a proxy anymore and will not delegate messages to `Node B`. As a

consequence, we need to extend the migration protocol such that we can ensure that an actor has processed messages that *might* be subject to break the desired causality relation.

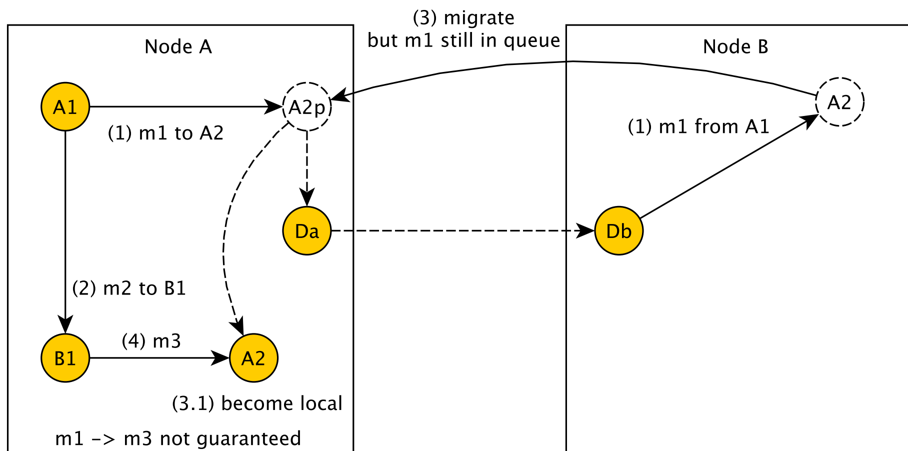


Figure 3.6: Causal Message Delivery - Broken

The protocol proposed in the following is not required to be executed for every case of migration, it is only necessary if migrated actors cause proxies to become local on the destination node. However, before migrating an actor, we cannot determine without sending any messages to the *Distribution Actor* of the destination node whether migration would cause proxies to become true actors.

The idea is to introduce a *confirmation-acknowledgment* round-trip to the migration mechanism in order to enforce that `A2` on Node `B` has processed m_2 before being migrated. Before the *Distribution Actor* `Db` migrates actor `A2` to `Node A`, it sends confirmation message containing the global IDs of the actors to be migrated to the destination node. The *Distribution Actor* `Da` receives this message and determines whether any of the global IDs match a proxy that exists on the node it manages. For each detected proxy, `Da` delegates the confirmation message (without any arguments) to the corresponding proxy. If a global ID cannot be matched to any proxy on that node, `Da` responds immediately with an acknowledgment message to `Node B`. If there is no proxy, then there cannot be any messages that have to be delegated to the actor which we plan to migrate. Hence, causal message delivery remains guaranteed without any extra efforts.

Actors for which a proxy exists on the destination node are not migrated until acknowledged. Once the confirmation message is processed by a proxy, it sends an acknowledgment message containing its own global ID to `Node B`. Furthermore, we prevent the proxy from delegating any messages to remote nodes from this point onwards. The *Distribution Actor* `Db` eventually receives the acknowledgment message and sends it (without any arguments) to the corresponding true actor of the same global ID. Once the true actor processes this acknowledgment message, it signals `Db` that it is ready for migration. Since messages are causal between the nodes, `A2` will have processed message m_1 before being migrated. At the same time, `A2p` will not have delegated any messages to `Node B`. Messages continue being processed once `A2` arrives at `Node A` and the local proxy becomes a true actor. Causal message delivery is guaranteed, as shown in Equation 3.5.

We do not have to consider `B1` sending m_3 to the proxy or to the true actor after migration. The important part of the protocol is to ensure that `A2` executed the state transition resulting from having processed m_2. Once the migrated actor is deserialized at the destination node, the proxy will have the same state as `A2` had on Node `B` at the time of migration. The fact whether `B1` sends m_3 to the proxy `A2p` or to the migrated true actor is not relevant for causality.

`A2` must have received *m1* before migration.

$$
\begin{aligned}
&A_1, N_A - (m_1) \rightarrow A_2, N_B \\
&A_1, N_A - (m_2) \rightarrow B_1, N_A \\
&N_B - (conf(A_2)) \rightarrow N_A; B_1, N_A - (m_3) \rightarrow A2, N_A \\
&D_a, N_A - (conf) \rightarrow A2p, N_A \\
&A2p, N_A - (ack(A_2)) \rightarrow A2, N_B \\
&\text{– migrate } A_2 \text{ to } N_A \text{ when ack received –}
\end{aligned}
\tag{3.5}
$$

Causality is not only a concern for actor migration but also for deserializing references to actors contained within remote messages. Hence, before discussing a more elaborate scheduler implementation, we will address the problem of handling actor references, provide a discussion on how serialization and deserialization is invoked and whether we can allow multiple actors or messages to be serialized and deserialized in parallel.

3.5.3 Handling Actor References and Actor Identity Comparison

Any actor may hold references to other actors as data members. Moreover, actors could also be referenced within any application message. Since a programmer using *Pony* is not aware of any topological information or the degree of distribution, it is necessary to provide a mechanism that is able to handle references to actors in the distributed context. The problem is that a receiving actor might be scheduled on a machine where references contained within any application message are not valid. In a concurrent setting, a reference is valid on a node, if it (the reference) points to an actor structure that has not been garbage collected. The same applies for migrating an actor which holds references to other actors as data members.

Migrating referenced actors to a the node of the actor containing the references is not always possible, because a single actor might be referenced by many other actors on different nodes. Furthermore, this approach would interfere adversely with the workload distribution scheme. The goal is to provide a mechanism that ensures consistency and delivers messages to the correct destinations.

The reader is encouraged to stop here for a while and think about the complexity of this problem. Not only is it a matter of handling references correctly, but also that the solution needs to consider issues related to routing, serialization and deserialization, causality and garbage collection.

In the following we will discuss how the *Distribution Actor* serializes and deserializes actor references. Furthermore, handling actor references needs to be designed with causality in mind. We first discuss the basic mechanism and then give an example that there is more to it in order to maintain causal message delivery.

Whenever a reference to an actor is sent to a remote node, we cannot be sure whether the reference is valid on the target machine or not. Note that the meaning of a *valid* reference changes in the context of *Distributed Pony*. We consider a reference to be valid on a node, if the same actor to which the reference on the origin node pointed to is located on the receiving node (which is possible due to *Pony's* proxy mechanism).

Encountering a reference to an actor during serialization and deserialization poses several challenges to the runtime system. Pointers to actors are serialized similar to the approach used for migrating actor structures, but ignoring the data part of the referenced actor. Instead, it is only required to send the type ID as well as the global ID of an actor to a remote node. Figure 3.7 illustrates the process of actor reference deserialization. `A1`, `A2` and `B1` are application actors, `A1p` and `B1p` actor proxies. `Da` and `Db` are *Distribution Actors*.
Upon deserialization, a proxy of the same type is created and the message (or actor structure) is given a reference to the created proxy. The *Distribution Actor* uses the global ID to send messages delegated via that proxy to the correct origin node, where the corresponding true actor is located. The *Distribution Actor* on the origin node is not required to maintain any information about having

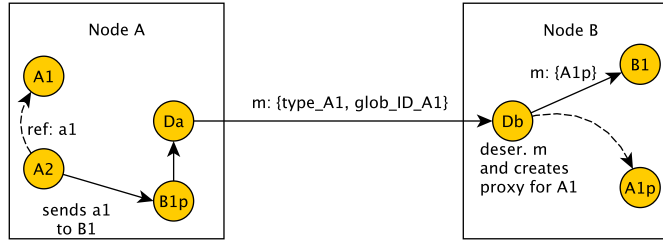


Figure 3.7: Serialization and Deserialization of Actor References

handed out references (except for garbage collection). The corresponding target actor for delegate messages can be found using the memory address contained within the global ID with which the remote proxy was associated upon deserialization. This is different from migration, where the memory address of the migrated actor on the new node is unknown at the time of serialization.

The use of global IDs is necessary to determine the identity of an actor, because a receiving node needs to check whether a proxy for a specific actor already exists. We must not ever have multiple proxies on the same machine for the same destination actor, because this would break the causality property of *Pony*, as shown in Figure 3.8 (distribution actors are omitted). Consider two actors `A` and `B`, both holding a reference to the same remote actor, but using different proxies. Actor `A` sends a message to `C` (via the proxy `P1`) *before* sending a message to `B`. In response to the message from `A`, `B` sends a message to `C` (via `P2`). Since message 1 precedes 2, which causally effects 3, we need to ensure that `C` receives 1 before 3.

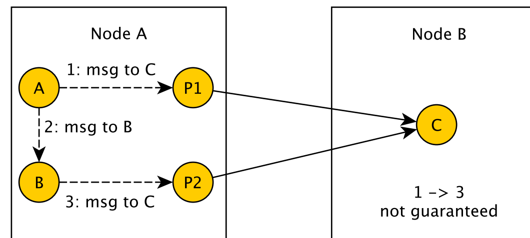


Figure 3.8: Multiple Proxies break Causality

Causality does not hold if multiple proxies for the same actor are located on the same node. The reason for this is that causality is not only broken in the distributed context, but also on `Node A` itself. There is no guarantee that the *Distribution Actor* on `Node A` receives message 1 before message 3, because the order in which the two proxies delegate the message is unknown. As a consequence, the order in which the *Distribution Actor* writes the two messages to the network channel is arbitrary. Hence, the global ID of an actor is used to determine whether a proxy is already available. Instead of creating a new proxy, references are simply replaced with the existent proxy upon deserialization.

From a language level perspective, the mechanism described above implicitly supports conditional code for *actor identity comparison*, because the runtime system guarantees that a node either inhabits a true actor or exactly one proxy for any migrated actor. By the time the *Distribution Actor* has deserialized an incoming actor reference, the actual pointer within the received structure is replaced with the corresponding proxy. Independently from which node actor references are received, pointers to the same actor are always replaced with pointers to the corresponding proxy. Since an application actor is not aware of any distribution, identity comparison of actors does not need to be handled separately. An equivalent approach allows to compare the identity of objects, which we discuss in section 3.6.

The extent of this problem and repercussions on other parts of the distributed runtime was not fully captured at the beginning of this project. It was tempting to provide a serialization

implementation that allows to serialize and deserialize messages and actors in parallel (by simply sending messages like "Deserialize yourself!"). The problem described above requires to maintain an identity map to correctly delegate messages and track existent proxies on a specific node. The parallel approach depends on a thread-safe hash map implementation, which itself is not a problem. However, depending on the application, this map is subject to high contention and in the critical path of remote message passing. Thread synchronization is therefore undesirable. As a result, we decided that the *Distribution Actor* will have exclusive access to that map and therefore no thread synchronization is needed.

In later stages of the project it also became evident that other parts of the runtime system also depend on the serialization and serialization to be done by the *Distribution Actor*. For example, a similar approach as discussed above is also the basis for solving the problem of distributed object identity comparison, which we discuss in section 3.6. The importance of this decision also becomes clear when discussing an implementation of distributed garbage collection, which is optimized for keeping the messages to be sent over the network to a minimum (section 3.7).

Some questions still remained unanswered. For example, actors might get migrated multiple times, which would build up a chain of proxies. Although this is an interesting property, a chain of proxies is undesirable. Furthermore, actors holding a reference to a proxy could be migrated to a node to which the corresponding proxy points to. Hence, we need to break "loops" of proxy delegation. Both issues are discussed in section 3.5.5.

However, the knowledge built up until this point is sufficient to discuss the advanced work stealing heuristics of *Distributed Pony*.

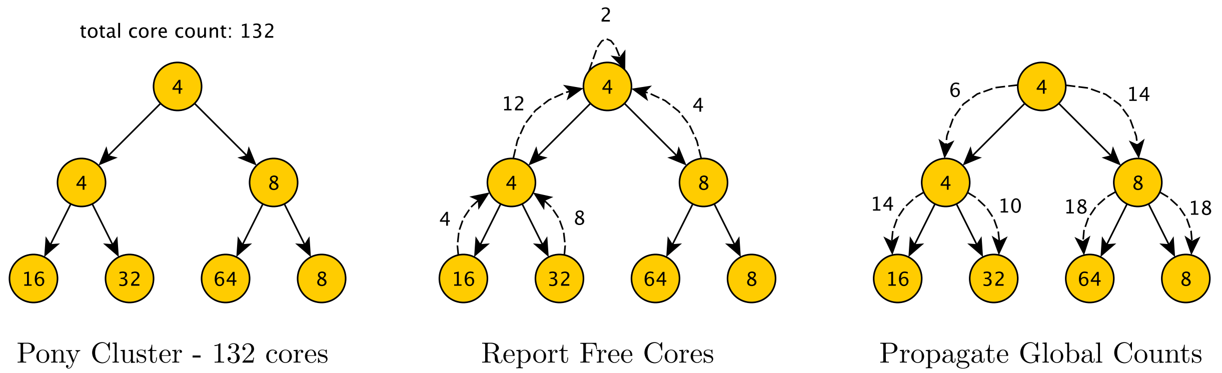
3.5.4 Hierarchical Work Stealing

We have established that the problem of distributed actor scheduling cannot be solved satisfactorily if only considering nodes attempting to steal work actively for themselves. A mechanism is required that detects to which part of the tree network actors should be migrated to. It is not sufficient to just delegate actors from the master node to participating *Pony* runtime processes down the tree. The scheme must be more adaptive, because arbitrarily many actors could be created and destroyed on any node during the lifetime of an application. Scheduling actors depends on the number of cores that idle in the *entire* system, not just a single machine. The fatal flaw of the basic scheme described earlier was that fully occupied intermediate nodes "partitioned" the network to some extent and caused nodes in subtrees to either starve from work or to be continuously overloaded. The distributed runtime scheduler for *Pony* however should be able to migrate actors from any node to any node when necessary.

Figure 3.9 illustrates a distributed work stealing scheme based on *push* and *pull*, where nodes actively advertise available cores *and* decide to delegate actors to other participants. The numbers in the nodes represent the total local core count, labels on the graph edges represent the reported free core count. Assume that the *Pony* cluster in the example has been running for a while. The proposed algorithm does not require to maintain a global view of the network.
The idea is to have each node to publish its free core count to its parent node. Thus, the knowledge of the amount of idle cores in the system propagates up the tree and eventually reaches the *master* node. Hence, each node in the system knows its own free core count and that of its children, transitively. The local free core count value of a node may oscillate frequently. In order to avoid that the system is flooded with core count messages, informing the parent node is triggered by a configurable kernel timer or by the receipt of an update message from any child node. Note that we do not propagate the free core count down the tree, because we would not know to which part of the tree we are publishing. This propagation is considered to be the *pull* part of the scheduling scheme, because advertising available cores is essentially equivalent to sending a request for actors to execute.

Whenever the master receives a free core count update message, it informs the other children about the number of idle cores available in the rest of the tree (i.e. via some path through the *master* and any of the other children). This is possible, because every node knows the core count of its children and any other nodes further down the (sub-) tree. For example, if the master node is

Figure 3.9: Advanced Work Stealing Heuristics



Pony Cluster - 132 cores   Report Free Cores   Propagate Global Counts

told from its right child that there are 4 cores idling in the right sub tree, it would inform the left child node (which is the parent of the left sub tree) that there are 6 cores available in a path via the master node (because the master itself has two unoccupied cores). Hence, the propagation of free core counts down the tree follows the calculation rule below, where i is the *i-th* child of node n and k the number of childes per node. The function *FreeCores* returns the *reported* free core count of a node.

$$\text{FreeCores} : \mathbb{N} \rightarrow \mathbb{N}$$
$$\text{CoreCount}(n, i) = \text{FreeCores}(n) + \sum_{a \in \{b | 0 \leq b \leq k\} - \{i\}} \text{FreeCores}(a); n, i, k \in \mathbb{N}$$

This process is repeated at every node except the leaf nodes. Hence, every *Pony* node maintains a triple consisting of the local free core count, the sum of the free core counts of all the descendants, as well as the idle cores in the rest of the tree reachable via some path through its parent node. Figure 3.10 shows the view each node maintains corresponding to the state of the system described in Figure 3.9.

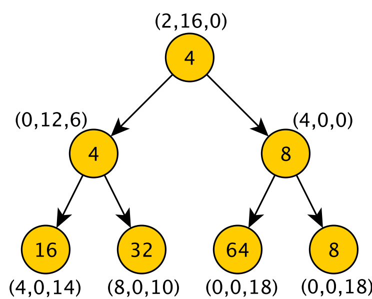


Figure 3.10: Core Count Configuration at each Node

Whenever a node propagates its free core count up the tree, it determines whether it might be necessary to send work to other nodes. If the children of a node reported idle cores, then their parent node *actively* migrates actors down the tree (if possible). This is the *push* part of *Distributed Pony's* scheduling mechanism.

Note that each node is able to determine whether it *might* have work to delegate by simply inspecting its own free core counter. We cannot entirely avoid unnecessary attempts to steal work, because there could be exactly one actor on each of the local queues. Incoming migrated actors are either deserialized and enqueued for processing or – if the free core count is zero – delegated

to some other node. In this case, the only overhead is re-transmission and not serialization or deserialization. How many actors are stolen depends on the node (actually: the path) chosen for delegation. Actors are only migrated up the tree if all child nodes (if any) report that they are fully occupied. This is sensible because of *data locality*. Remember that the proposed joining algorithm guarantees that a tree of *Ponies* is eventually balanced. As a consequence, the number of hops necessary in order to reach a node through a parent which is not used to capacity is potentially larger than a path via one of the child nodes.

One might come to the conclusion that a limitation of this scheme is that the workload distribution is not guaranteed to be uniform. If all nodes of a *Pony* cluster are used to capacity, no actors are migrated. Hence, the scheduler seems not to be optimized to deliver the shortest latency. The turn-around time of an actor might be higher than necessary, because a node might have substantially more actors to execute than others. However, the important point is that *work stealing* is about stealing *work* and not stealing *actors*. As a result, we do not need a solution to this problem, which could be to keep track of the number of live actors per node. Using this information, nodes could propagate their load factor additionally to the free core count. This enables each node to determine the occupancy of the sub trees through their children and the rest of the tree through their parent node. Migration could then also be based on detecting that the load factor of nodes within the tree is imbalanced.

The problem however is that no node knows the core count of every other node in the system, which makes it difficult to determine how many actors should be migrated, because the impact of migrating a single actor on the occupancy factor of a node is unknown. The relative speeds of the available processors might vary greatly, which also influences the turn-around time of an actor. Migrating actors from a fast machine to a slower machine based on the occupancy factor may therefore be exactly the wrong thing to do. For this reason, we believe that the proposed scheduling algorithm is a good approximation, because it is optimized for *throughput* and therefore also *latency*).

Migrating actors from one node to another causes the creation of a proxy to delegate messages. Since migration may happen multiple times, this process could build up a chain of proxies. Also, actors could be migrated back to nodes on which they have previously been or to nodes where a referenced actor is scheduled. In order to avoid unnecessary message delegation or serialization and deserialization overhead, these issues need to be considered by the migration mechanism. Solutions to these problems are discussed in the following.

3.5.5 Collapsing Actor Proxies

Technically, a distributed programming language runtime such as *Pony* does not necessarily require a prefix-based scheme to route messages from source to destination. In fact, the first prototype of *Distributed Pony* routed messages implicitly through building up chains of proxies. Assuming no failures of nodes, the migration mechanism described in section 3.5.1 is sound, such that the destination actor is eventually reached. A migrated actor might be scheduled to another machine again, such that a new proxy would be created that delegates messages to the new target and so on. Although sufficient for an initial showcase, this approach violates one of the desired properties we set ourselves for *Distributed Pony*, namely, that serialising and deserialising a message or actor should only happen once.

Not testing for a target actor to be a proxy would result in unnecessary overhead due to serialization and deserialization. The *Distribution Actor* would read the remote message from the socket buffer and simply delegate the message to the target actor (which might be a proxy). The message is deserialized but not evaluated. A target actor being a proxy would cause the message to be serialized again and sent to the node where the actor was migrated to next. This process could be repeated several times until the destination is reached.

In order to reduce the overhead to that of simply re-transmitting a message to the next hop, the *Distribution Actor* checks whether a target actor for a remote message is a proxy or not. This is achieved by checking the hash map maintained for delegation. If an entry is available, the node ID of the machine to which the actor was migrated to is extracted. The *Distribution Actor* sends

a re-targeting message to the origin from which the delegated message was received. This message contains the global ID of the corresponding proxy on the origin node as well as the new node ID. Using this information, the *Distribution Actor* at the origin node will update the hash map maintained for proxy delegation accordingly. Note that this does not mean that subsequent delegate message will not be routed via the node which sent the re-targeting message, but intermediate *Distribution Actors* can detect that the message is meant for another node, and would therefore forward it towards the destination without reading the message content, as explained in section 3.3.2. Also, one node might receive multiple re-targeting messages for the same proxy. The overhead of sending messages to remote actors is reduced to that of simply re-transmitting the message, instead of deserializing and serializing the message at each intermediate node.

The runtime system takes also care of unnecessary "loops" of message delegation. Consider an actor holding a reference to a proxy. If this actor is migrated to another node on which the actual counterpart of the referenced proxy is scheduled, it would be unnecessary to create a proxy for that reference and let it point back to the origin node. This would cause messages to be delegated in a loop. Instead, the *Distribution Actor* detects (using the global ID) that the referenced actor is actually on the same node and therefore replaces the reference with a pointer to the actual actor during deserialization accordingly.

Intermediate actor proxies can *not* be destroyed immediately after a re-targeting message has been sent, because there might be local actors holding a reference to any of the intermediate proxies. However, proxies are garbage collected just like any other actor in the system, and therefore destroyed when possible. Hence, *collapsing proxies* only avoids unnecessary evaluation of application messages, but does not necessarily reduce the number of proxies for a specific actor. There is no invariant that an actor has only one corresponding proxy from the perspective of the entire *Pony Cluster*. However, it is guaranteed that there is no more than one proxy for a any actor on each node and that a remote message is serialized and deserialized exactly once.

Interestingly, collapsing actor proxies and reducing the remote delegation overhead to that of re-transmission is exactly the reason why we had to implement a routing scheme. Building up a chain of proxies would guarantee that the destination is eventually reached, without maintaining *any* (explicit) routing information. However, serialisation and deserialisation is too expensive compared to determining the next hop towards a destination.

The ability to collapse proxies depends on the availability of the knowledge about which proxies exist on a particular node and to which original actor each of them corresponds to. Thus, the identity of an actor must be known in order to determine corresponding proxies. A similar form of identity is required due to a programming language feature of *Pony* (object identity comparison). The challenge in this context is not how to avoid unnecessary serialization or deserialization, but to provide a mechanism that is not massively slower compared to the concurrent setting. Hence, sending messages over the network is not an option in this case. The following section discusses a scheme to compare the identity of objects in a distributed context without sending any messages or changing the storage layout of *Pony* objects.

3.6 Distributed Object Identity Comparison

Object identity comparison is a very common feature among object-oriented programming languages like C++ [116] or Java [67]. The idea is to test whether two different variables point to the same memory location and, depending on the type, whether the two variables point to the same object. *Pony* supports identity comparison, too. In a concurrent context, this feature is easy to implement, because only the values of the memory addresses need to be compared. The type system guarantees that compared variables are of the same type. Contrary to a single machine, memory addresses are not unique across a network of computers. So, checking for identity using these addresses is ambiguous. Not only is this feature important for *Distributed Pony* on a language level, but also important for serialization and deserialization (in combination with the type system to be developed for *Pony*).

The proposed actor proxy mechanism together with handling references to actors, which we

discussed in section 3.5, required identity comparison for actors. Neglecting the ability to send and receive messages and maintaining a local heap, actors are quite similar compared to objects. It stands to reason that we can therefore re-use the *hash map* implementation in combination with the concept of *Pony global IDs* for distributed object identity comparison.

For efficiency reasons, we want to provide a mechanism without changing the layout of objects, because this would lead to substantial overhead if many objects exist during the lifetime of an application. Not only would this affect the performance of *Concurrent Pony*, but also negatively influence serialization, deserialization and transmission of remote messages.

Hence, whenever a complex type is serialized, we do not only write the actual data, but also the corresponding global ID, which consists of the node ID and the memory address of the object. Upon deserialization, a *Distribution Actor* allocates the necessary memory space for the object and maintains a mapping of the object's memory location on the receiving node and the extracted global ID. Therefore, in *Distributed Pony*, two objects are identical if their global IDs (maintained by the *Distribution Actor*) are the same.

The main requirement for *Distributed Pony's* identity comparison mechanism is that there should be no performance difference compared to the concurrent setting. Since this issue is technically similar to the one discussed for handling actor references, efficiently comparing the identity of objects requires to implement deserialization within the *Distribution Actor*. Allowing parallel deserialization would require to synchronize the access to the hash map data structure, which is undesirable and in this case worse than deserializing on a single core [6]. Figure 3.11 shows an example situation to explain distributed identity comparison. For simplicity, proxies are omitted in the diagram provided below:

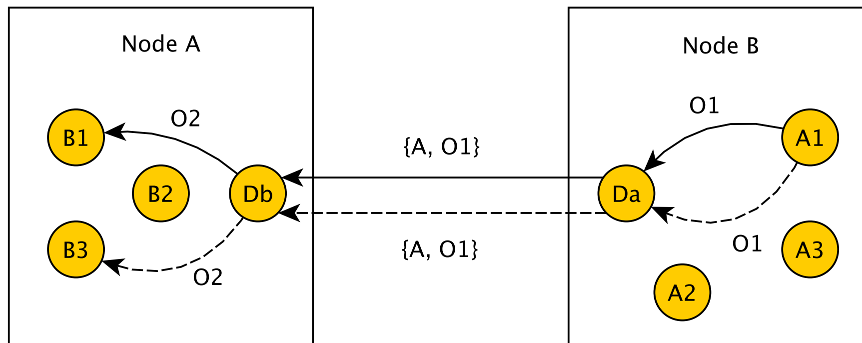


Figure 3.11: Pony Identity Comparison

Consider a remote message, which contain a reference to an object `O1`, sent from Node `A` and actor `A1` to an actor `B1` which is located on Node `B`. The *Distribution Actor* `Da` serializes the message containing `O1` as `{globID_A_O1, O1}`. Similar to the global ID discussed for actors, `globID_A_O1` contains the memory address `O1` has on A as well as the node ID of `A`. `A` is the node where the object `O1` was created. *Distribution Actors* on other nodes re-use `globID_A_O1` when serialsing a structure that holds a reference to `O1`. Upon receipt of such a message, Node `B`'s actor `Db` deserializes the message and allocates the necessary space for `O1` at address `O2`. Moreover, `Db` maintains a mapping in its identity map of `globID_A_O1 -> O2`.

Whenever `O1` is sent within a remote message from some actor to any actor on Node `B` again, the *Distribution Actor* at `B` replaces `O1` in the message with a reference to `O2`. Identity comparison on Node `B` can then safely be done using the memory location of `O2` (like in a concurrent setting). Remarkably, we have a mechanism that allows for comparing the identity of objects in a distributed context using the *same* binary code as in the concurrent setting, because the code within an actor does not have to be touched. There is no overhead or compiler code generation involved. Furthermore, providing thread-safe access to the identity map is not required, because this data structure is private to the *Distribution Actor*.

It is a consequence of the actor paradigm that the implementation of *Concurrent Pony* re-

mains unchanged to a large extent. Distribution is basically concealed within the discussed actor. Hence, tuning the performance of *Distributed Pony* is equivalent to tuning the implementation and scheduling of the *Distribution Actor*. As a result, there is no need to trade off competing factors between the implementation of concurrent and distributed *Pony*, because both concerns are loosely coupled.

A variation of the discussed identity map is also useful for distributed garbage collection. The goal is to keep the number of messages to be sent over the network for garbage collection to a minimum. The following sections discuss *Pony's* distributed garbage collection mechanism, which – independently of how often an object is sent to a node or to how many remote actors – requires exactly *one* network message to signal the origin node that all actors on the corresponding remote node have released their interest in a specific object. Garbage collection is not only about reference counting, but also about detecting reference cycles. Hence, section 3.7.3 and 3.7.4 discuss a basic centralized and a more sophisticated distributed implementation for cycle detection.

3.7 Distributed Garbage Collection

A feature unique to *Pony*, compared to other actor-based language runtimes, is fully concurrent garbage collection of actors [30]. *Pony's* garbage collector needs to be aware of actor references handed out to remote nodes as well as actors that have been migrated. This is necessary because we need to avoid that blocked actors are collected although there might still be references remotely. We have given a short overview of *Pony's concurrent* garbage collection scheme in chapter 2. In the following, we propose an algorithm for fully concurrent garbage collection of actors in a distributed context. Note that we could technically enable the already available garbage collector to work in a distributed context without much effort, but for the reasons discussed in section 3.7.3 we decided to develop a more elaborate scheme.

Pony implements both passive object and active object collection. We refer to passive object collection when talking about any type which is not primitive nor an actor (i.e. normal objects as you can find them in object-oriented languages) and to active object (or actor) collection when talking about collecting actors.

3.7.1 Deferred Reference Counting and Passive Object Collection

Pony implements deferred reference counting, which means that the reference counts of object and actors are only updated when messages are sent and received or when an actor executes a local garbage collection cycle, as opposed to updating reference counts upon operations like pointer assignment and so on.

For garbage collection purposes, every actor maintains two maps, an *external reference* map and a *foreign reference* count map. The *external reference* map contains every other actor or object (which is allocated on another actors heap) to which an actor may hold a reference to. At the same time, this map contains information about the owner of every referenced object, such that we are able to find the actor on whose heap the referenced object was allocated. Note that it is possible that actors hold *direct* references to other actors' heap due to the safe zero-copy messaging semantics we have mentioned in chapter 2.

The purpose of this map is to maintain references received within a message. Actors being created by other actors are also added to this map. An actor's external set is eventually compacted, and unreachable references are removed as soon as an actors heap becomes swept during execution of a local garbage collection cycle.

A second data structure is necessary to maintain a map of objects allocated on an actors heap which are accessible to other actors. We refer to it as the *foreign reference* count map. This data structure maps a local object to its foreign reference count. This means, that an objects reference count applies to references that many other actors may hold on their heap. The reference count is said to be deferred, because we do not increment it upon pointer assignment or similar operations, but instead manage reference counts more lazily when sending and receiving messages or executing local garbage collection cycles.

Whenever a local object is sent in a message, we increment its reference count immediately. This is a consequence of the asynchronous nature of the message-passing paradigm. Actors may execute garbage collection cycles while messages are pending in their mailbox. We need to ensure that an object is protected from being garbage collected if contained within possibly unprocessed messages.

Actors can also hand out references to objects that are allocated on another actor's heap. Instead of synchronizing the access to the externally owned object, we send a reference count increment message to the owning actor. Actors add received objects to their external set. If a received object is already present in this set, we send a reference count decrement message in order to correct the "protective" increment of the sending actor. This is a good approach, because an actor is not required to maintain a global view of the actors and objects a receiving actor holds references to.

An actor executing a local garbage collection cycle causes every reference in the external set to be marked as reachable or not. This is implemented by using the discussed *trace* mechanism in mode `TRACE_MARK`. Unreachable references can be removed from the external set and decrement messages are sent to the owning actors. There is no invariant that a reference count is equivalent to the number of referrers, because there might always be unprocessed increment or decrement messages. However, it is only necessary to ensure that a reference count value is greater than zero when an object is reachable by some actor. An object can be deallocated if its reference count value is zero, in which case it not reachable by any actor and there are no messages pending in any message queue containing that object.

Since reference counting in *Pony* is based on message-passing, we can re-use the same mechanism in a distributed context. The mechanism discussed for garbage collecting passive objects is the basic mechanism for actor collection. Therefore, we give a detailed example of distributed reference counting for both cases in the next section.

3.7.2 Active Object or Actor Collection

In the context of reference counting only, actors and objects are treated in the same way. However, in the distributed context, our goal is to keep the number of increment and decrement messages sent across the network to a minimum. The proposed actor proxy mechanism discussed in section 3.5.1 allows us to do this conveniently. Figure 3.12 shows an example scenario of an actor `A1` sending a reference to itself to an actor `B2` which is located on another node. Assume that `Node B` has never heard of actor `A1` through previous messages or migration. Also, `A1` holds a valid reference to a proxy for `B2`.

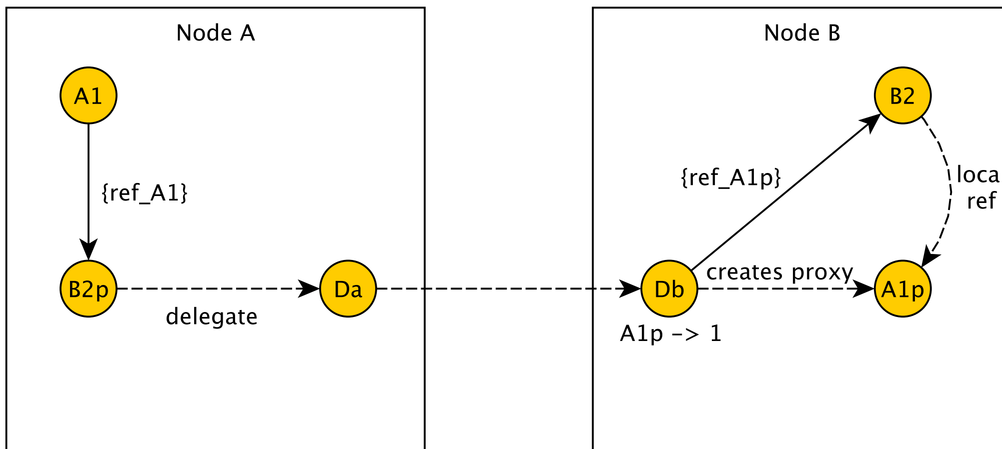


Figure 3.12: Distributed Actor Reference Counting

`A1` increments its own reference count before sending the message to `B2`. The message is eventually delivered to the remote node via the local proxy `B2p` and the *Distribution Actor* `Da`. `Db` deserial-

izes the message and exchanges the reference with a pointer to a newly created proxy `A1p`. The *Distribution Actor* on Node `B` sends a reference count increment message to `A1p` when delegating the message to `B2`. Also, `Db` maintains a mapping of the proxy to the number of references that have been received for a particular true actor (in this example `A1->1`). References to `A1` might be received many times (and from different remote actors and nodes), which causes the reference count of `A1p` to be incremented as well as the counter held by the `Db` to be updated. The proxy `A1p` is eventually garbage collected. The `Db` needs to be informed about the proxy being deallocated and in response sends a decrement message to the node where the true actor `A1` is present. This decrement message contains the number references to `A1` the `Db` has deserialized (e.g. `DEC(20)`). By doing so, we can decrement the reference count of `A1` accordingly, releasing all references held by a particular remote node with a single message. Again, the global ID is used to determine the target actor of the decrement message.

A similar approach can be used for distributed reference counting of objects, as shown in Figure 3.13. The two approaches are equivalent except for the fact that for object reference counting no actor proxies are involved. The important part is that a *Distribution Actor* takes over the ownership for objects received from remote nodes. The role of ownership from the perspective of other actors on the same node is ensured implicitly because the *Distribution Actor* deserializes any received objects and allocates them on its own heap. This is necessary, because we need to be able to have a recipient of decrement messages from other actors. A proxy for the actual owner of an object may not be present on a every remote node. The owning *Distribution Actor* of remote copies of the object (in the example below `Db`), can sent the decrement message to `Node A` as soon as the remote copy was garbage collected locally.

In order to deliver the decrement message from a *Distribution Actor* of a remote node for an object, we need to be able to determine the owning actor. This is possible without requiring *Distribution Actors* to maintain an additional data structure. The owner of an object can be determined based on its memory address (included in the global ID). The base address can be used to lookup the owning actor in a page map, which is available for garbage collection purposes.

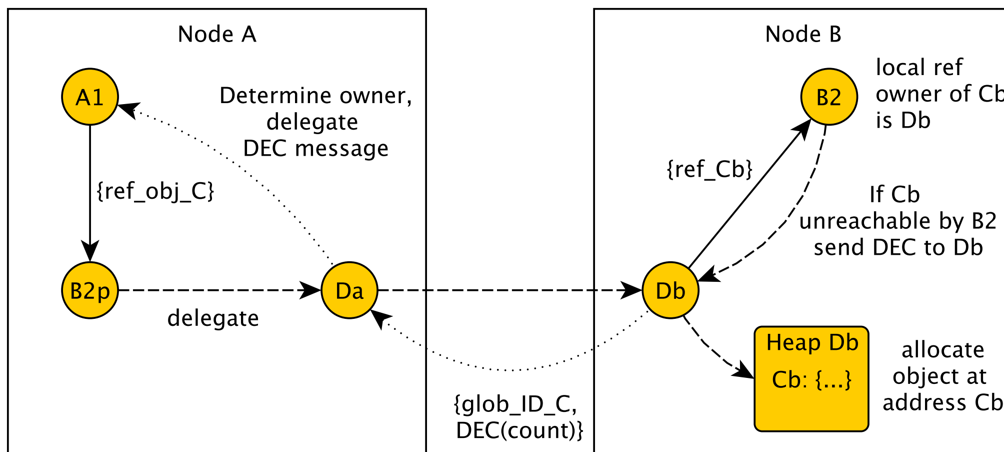


Figure 3.13: Distributed Object Reference Counting

The benefit of this scheme is that independent of the amount of references that have been sent out to a remote node, we are only required to send exactly one decrement message for each object or actor which was referenced remotely instead of flooding the system with reference count messages.

Actors may reference other actors. Cyclic garbage cannot be collected using reference counting alone, because the reference count values of actors forming a cycle never reaches zero. We have previously mentioned that *Pony* implements cycle detection based on an actor that is able to determine true cycles based on the topology reported from blocked actors. Once ensured that the perceived cycle is a true cycle and that the information reported is in sync with that of all other actors in the system, the cycle detector allows to collect dead actors involved in a cycle. In a

distributed context, detecting cycles seems a bit more complex, because there is not necessarily a global view of the entire actor topology locally available at every node. Although local actors technically reference the local proxy of every actor that is located remotely, we do not know the state of the true actor on a remote machine. Furthermore, we have no information about the set of remote actors that might have references to actors participating in a local cycle. This also means that a cycle can consist of actors that are all located on a different node. Detecting cyclic garbage reliably in actor-based systems is difficult, because the view of actors participating in a cycle as well as that of the cycle detector itself may be out of sync.

Note that cycle detection for object references is not necessary in the context of *Pony*, because actors (and not objects) are responsible for sending reference count increment and decrement messages for both object and actor references.

In the following, we present a basic approach and an optimized algorithm for detecting cyclic garbage in distributed and actor-based systems. Our motivation for a more advanced approach was to leverage the characteristics of the underlying network topology as well as to reduce the message overhead to a minimum. At the same time, we want to avoid that a particular node (or cycle detector) becomes a bottleneck, which is important for the performance of *Distributed Pony* running on large-scale computer clusters.

3.7.3 Centralized Cycle Detector

In order to be able to collect cyclic garbage, *Pony* introduced a detection mechanism implemented through an actor (the cycle detector) which requires causal message delivery, as we discussed briefly in chapter 2. An actor is said to be *dead* if it is blocked an all actors that reference it are blocked, transitively [30]. Whenever an actor becomes blocked it sends a message to the cycle detector containing itself, its current reference count and the set of actors it references (taken from its *external reference* map). This information is called the *perceived* actor state based on which the cycle detector establishes *perceived* cycles by building sets of blocked actors that form a cycle. The difficult part is that, once the cycle detector receives a message containing the view an actor has on its own topology, this information may be already out of sync, because the sending actor may have become unblocked again.

Consequently, we need to inform the cycle detector about this view change. Hence, when an actor becomes unblocked it sends a message to the cycle detector containing itself, such that we can remove this actor from the set of blocked actors. A perceived cycle containing an unblocked actor cannot be a true cycle and therefore should not be garbage collected. As a result, upon receipt of an unblock message, the cycle detector cancels any perceived cycle containing the unblocked actor.

The cycle detector having detected a perceived cycle must ensure that its view on the actor topology is in sync with the view of all actors forming that cycle. This is where causal message delivery comes into play. The cycle detector sends a confirmation message (including a token that identifies the perceived cycle) to all actors of the cycle. Actors receiving that message always acknowledge it - independent of their own state. If the cycle detector does not receive an unblock message before receiving the acknowledgment, then the decision based on which the cycle was detected must have been a topology that agrees with that of the confirming actor (because it did not unblock in between and messages are causal). If all actors in the perceived cycle can be confirmed, a true cycle is detected and all participating actors can be collected.

In a distributed setting, detecting cycles is more complex, because actors forming a cycle could be located on different nodes. This means that the information necessary for determining a true cycle is distributed. One solution to this problem could be to have a central cycle detector scheduled on the master node. All nodes delegate any of the messages discussed above to the master node. By doing so, the central cycle detector obtains a *global* view on the actor topology across all nodes can therefore detect true cycles. The only change necessary is to tell the cycle detector the global ID of an actor, such that confirmation message and deallocation can be delegated to the correct actor and node on which it is located.

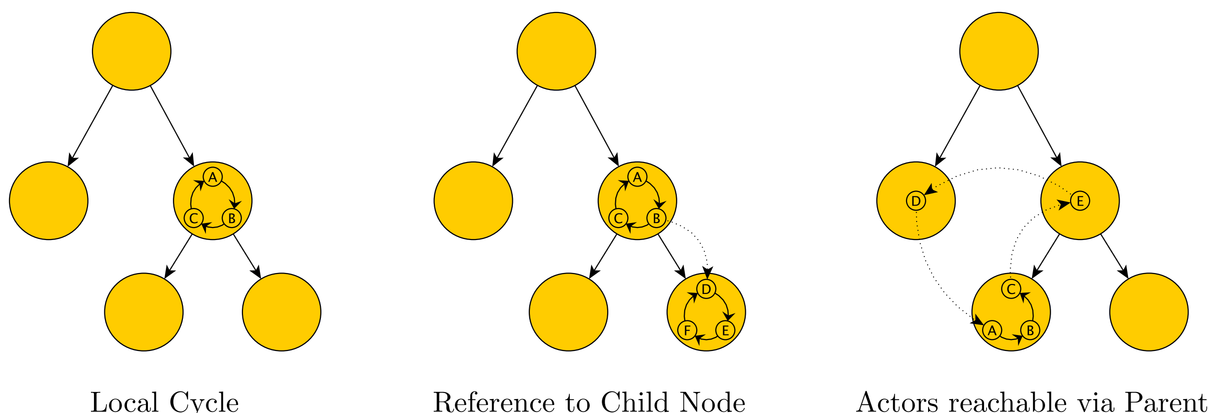
The problem with a centralized approach is that many messages are send across the network and a single cycle detector may become a bottleneck, especially when running on a large-scale computer

cluster. In fact, sending information up the tree to the root node is unnecessary in many cases. In the following, we propose a hierarchical cycle detection algorithm that minimizes the amount of messages to be sent and attempts to avoid that information is not delegated to nodes up the tree hierarchy unnecessarily. Furthermore, we want to have fully *concurrent* garbage collection of actors in a distributed setting, i.e. cycles that are detected locally and only contain actors on the same machine should be collected using only the cycle detector on that node.

3.7.4 Hierarchical Cycle Detection

Instead of employing a central cycle detector on the master node, we want to have a cycle detector available on every node in a cluster of *Ponies*. We shall illustrate our hierarchical cycle detection algorithm based on three examples as shown in Figure 3.14.

Figure 3.14: Distributed Cycle Detection



If all dead actors participating in a cycle are local to a node and none of these actors are referenced from other actors located on a remote node, we can safely collect the cycle locally without sending any messages across the network. This is the equivalent situation as we have discussed for a single-node environment. Hence, collecting local cycles is possible with no overhead in *Distributed Pony* compared to the concurrent setting. Note that a centralized cycle detector would have required to send messages up to the root node.

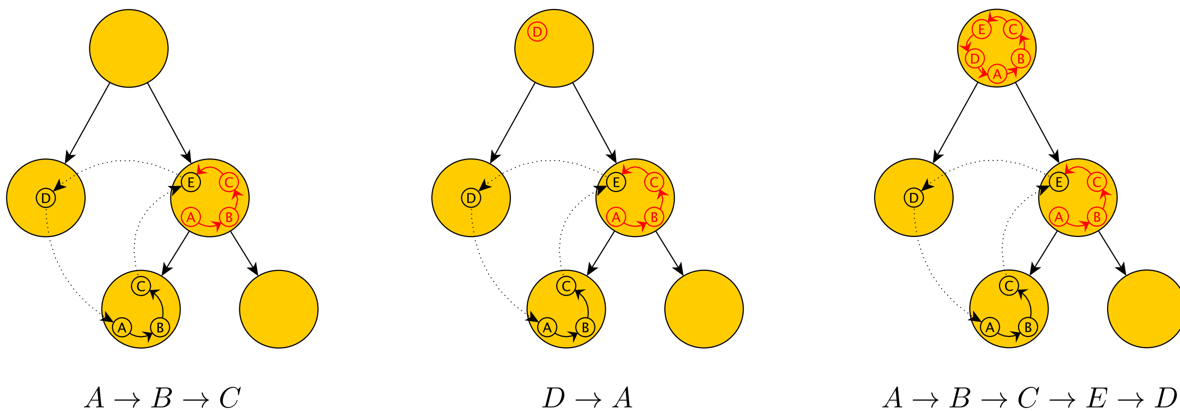
A more complicated situation comes up when actors in a perceived cycle hold references to actors reachable via a child node. In the example given above, we may not allow the cycle in the right leaf node to be collected before the cycle containing the remote actor B that holds a reference to D and the proxy for D at the node of B is collected. So, the local cycle detector running on the right leaf node needs to hold on to that cycle. The necessary information is implicitly available due to our reference count implementation. Every actor in a perceived cycle is possibly blocked, which means that the reference count of every actor in the perceived cycle is equivalent to the number of referees. Hence, we can use this reference count invariant to determine whether there are any remote references. In the example above, the reference count of D is higher than the number of referees that the cycle detector can be aware of, in which case there must be unreleased references from other nodes.

Actor B technically holds a reference to a proxy Dp for actor D. Eventually, the perceived cycle of which B is part of is verified to be a true cycle and can be collected. The proxy is also eventually collected, and in response to that the *Distribution Actor* of the corresponding node sends a reference count decrement message to the node of D. Once the decrement message is processed by the *Distribution Actor* of the right leaf node, the cycle detector is able to establish that there are no pending remote references. From point onwards the local cycle detection protocol is used. Confirmation messages are sent to the participating actors and once all acknowledgments are received, the cycle can be collected. Only one reference count decrement message was necessary to collect both cycles.

The most difficult situation is when actors forming a cycle are located on different nodes, in which case local actors are uncollectable. However, we can provide an efficient algorithm that leverages properties of the underlying tree network topology. Remember that when an actor becomes blocked, it sends a message to the (local) cycle detector containing itself and the set of actors it references. This enables the cycle detector to check whether any of the reported outgoing references are pointing to an actor proxy. In a tree network topology, a cycle can only exist between actors located on different nodes, if at least one actor (of a possible cycle) holds a reference to a proxy for an actor that is reachable via a parent node. In the example above, this is the case for the actors `C`, `D` and `E`. We consider an actor to be reachable via a parent node, if sending a message to it requires to route through a parent node of the node where the sender is located.

In such a case, the local cycle detector promotes the collected information to the parent node, including the global IDs of the participating actors, as shown in Figure 3.15. Eventually, the collected information can be merged at the earliest common ancestor of all nodes holding actors participating in a possible cycle. In the given example, the earliest common ancestor of the nodes in observation is the root node of the shown (sub-) tree. Actor `A` holds a reference to actor `E`. Hence, we promote the local cycle detectors view of the topology of `A` to the parent node. This allows the cycle detector of the node containing `E` to extend its view on the topology. `D` holds a reference to `A`, which causes $D \rightarrow A$ to be reported to the parent node. `E` holds a reference to `D`, so the cycle detector reports its (already extended) view of the topology up the tree. Eventually, the partial actor graphs can be merged to a graph where all contained actors are either located on the current node or are reachable via some of the child nodes. Note that any unblock messages for actors contained within partial graphs that have been promoted up the tree also need to be delegated to the “overseeing” cycle detector.

Figure 3.15: Hierarchical Cycle Detection



The cycle detector which has obtained the merged view of a perceived cycle executes exactly the same protocol as the one previously discussed. A confirmation message is sent to the actors contained with the perceived cycle. Note that we can determine the destination node for each of the actors based on their global ID. As previously, actors acknowledge the confirmation message. Since message delivery is causal (see chapter 4), a true cycle is determined if no unblock message has been delegated to the “overseeing” cycle detector between its confirmation message and all acknowledgments from actors contained within the perceived cycle. If the perceived cycle can be confirmed, the “overseeing” cycle detector sends messages to the local cycle detectors that the actors located on their node which are part of the distributed cycle can be collected safely. The set of actors to be collected can be determined using a token identifier.

If the perceived cycle cannot be verified as true cycle, because an unblock message from any of the contained actors was received before the acknowledge, we need to be careful about what information to discard at the “overseeing” detector and what information to keep. We want to avoid that local cycle detectors have to report their view on the topology again, even if their part of the graph has not changed. Hence, the “overseeing” detector only removes those actors from the

perceived cycle that sent an unblock message. For example, if an unblock message of D is delegated to its parent node, we only remove D from the perceived cycle instead of canceling the entire cycle. We keep the rest of the information, because the remaining actors are located on different nodes and the view the "overseeing" detector maintains has nothing to do with the topology of actors on its node. As a consequence, if D becomes blocked again, we need to be able to reconstruct the merged view on the topology again. In a distributed setting, we would be required to ask for the view of the local cycle detectors again, which is unnecessary message overhead. Instead, if D becomes blocked again, we simply restart the confirmation protocol.

We have presented an algorithm for distributed cycle detection that avoids a single cycle detector to become a bottleneck. Instead, our algorithm detects cycles as far down in the tree topology as possible. Furthermore, local cycles can still be detected and collected on every node fully concurrently. Network messages are only sent if an actors hold references to remote actors reachable via a parent node as well as for the confirmation protocol necessary for synchronizing different views on the topology. Furthermore, the algorithm guarantees that the paths for sending the messages necessary for the garbage collection protocol are as short as possible, which reduces retransmission overhead. In future work, a soundness and completeness proof for the proposed distributed cycle detection algorithm shall be provided.

Two language features available in *Concurrent Pony* are left to be discussed in the distributed context, *termination* and *causality*. In a single-node configuration, termination can be detected based on quiescence. If all actors in a system are blocked, then there cannot be any messages left to be sent. However, detecting that all actors are blocked in distributed system is more complex.

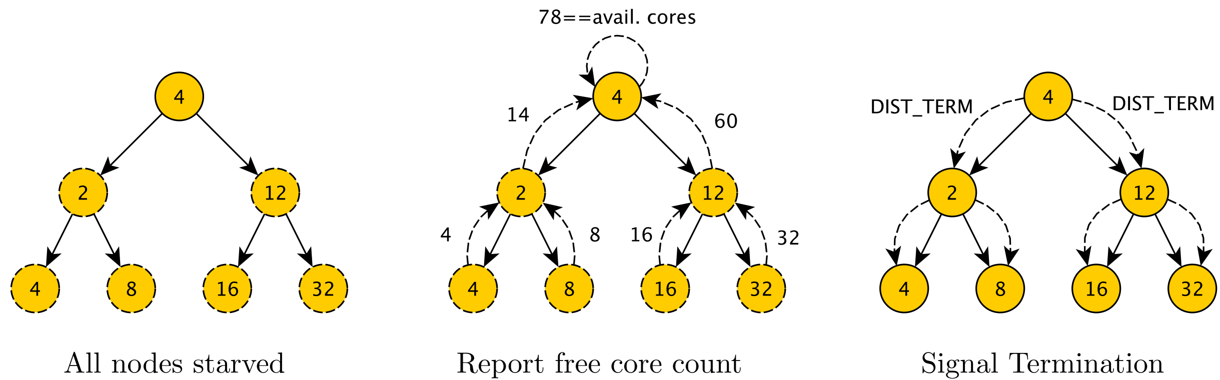
3.8 Termination

We have no real notion of quiescence in a distributed context, because a local node may just have run out of work and needs to steal actors from remote machines (which themselves may have no work to provide). At the same time, there may theoretically always be a message to be delivered from a remote node. Hence, we need to "redefine" what quiescence means in a distributed network of *Ponies*. The solution for distributed termination was more difficult than anticipated. Our initial solution was soon found to be broken. To illustrate the development process of the termination algorithm, we will explain our approach incrementally, and first discuss the broken mechanism in order to motivate a more complex but correct algorithm.

For our first attempt, we have slightly changed the algorithm from section 3.3.2 for joining a new slave node to the system. Whenever a node connects to the *master* attempting to join a network of *Ponies*, it includes its local core count within the join request message. By doing so, the *master* node can keep track of the total core count available within the *Pony* cluster it is part of. We can use this knowledge in order to determine whether there are any cores left that are occupied or not. As shown in Figure 3.16, each node (except the master) in the system reports the *free* core count to its parent. This information is delegated in an aggregated manner, such that every node reports the free core count of itself and its children, transitively. Eventually, these notification messages will "bubble" up to the master node. The idea was to use the total free core count to detect whether all cores have starved from work. If so, there cannot be any messages left - because all actors must be blocked. The *master* could respond with a termination message (DIST_TERM). Every node – including the *master* – can safely terminate immediately after a message of type DIST_TERM has been received and delegated to all children. The result would be a distributed termination algorithm with a complexity of $O(\log n)$ (time and space).

However, the sampling rate at which nodes report their free core count is not necessarily equivalent to the rate at which busy cores become idle. Since we want to avoid flooding the network with messages, we set a simple kernel-timer to report core counts and only send notifications if the value changed compared to last time. The latency involved with this approach is not problematic, because termination is not performance critical. Note that the mechanism we chose for scheduling the *Distribution Actor* interferes with counting the number of free cores, because the *Distribution Actor* keeps on polling for network I/O. As a result, a node is never entirely free of work. This

Figure 3.16: Distributed Termination (Broken)



All nodes starved   Report free core count   Signal Termination

would break the termination algorithm described above, because the entire set of processor cores of a *Pony Cluster* would never be reported as idle. Therefore, it is necessary to detect whether the *Distribution Actor* is the only actor being unblocked on a node. This is difficult, because we cannot efficiently walk through the scheduling queues and check their state. Even if were able to determine that the *Distribution Actor* is the only actor remaining on the last non-idle core of a machine, there could technically always be a message in the network I/O buffer which has not been processed at the time of detecting free cores. The problem is, that the view a node had on its free core count may be out of sync at the time all messages have bubbled up to the master node. We had encountered a similar problem when discussing cycle detection for actor garbage collection. The key to the solution in the context of garbage collection was causality. We attempt to develop a solution for distributed termination based on the same protocol.

The idea is to establish a central termination actor – called the *termination detector* – at the master node. Whenever a node in the network detects that it has starved from work, it sends a "*starvation*" message to the central termination detector. Similarly, if a node gets work to do again (some actor becomes unblocked), it sends a "*busy*" message to the termination detector. If the termination detector has received starvation messages from all nodes, it sends a confirmation message including a unique token identifying the current termination attempt to all nodes within a *Pony* cluster. In fact, it only sends this message to its direct children, which in turn delegate this message to their children and so on. Any node (or technically *Distribution Actor*) receiving this confirmation message responds with an acknowledgment message including the same token, independently of whether the machine is busy or not. Assuming that message delivery is causal in a tree network topology (see chapter 4) and the central termination detector receives acknowledgment messages from all nodes for the corresponding termination attempt, without having received any "*busy*" messages in between (in which case the views of all nodes must be in sync), it can safely send the previously mentioned `DIST_TERM` message to its children. Any child node receiving this message delegates it further down the tree and then terminates. Eventually, the entire system terminates. Note that terminating a single *Pony* runtime process is trivial, because we only need to destroy the *Distribution Actor*, which must have been the last remaining unblocked actor on the system. Once destroyed, each runtime process is terminated using the quiescence scheme as discussed for *Concurrent Pony*.

This algorithm can be optimized at various points. For example, always sending a message to the central termination detector in any case is unnecessary. Instead, it is sensible to only send starvation messages to the parent node. If the receiving parent is busy it may hold on to that message up to the point it receives a busy message from the node having sent the previous starvation message. If the parent node becomes idle, it may delegate an extended starvation message including itself and all children having reported to be starved to its parent. In large computer clusters this can be sensible, because we can reduce the number of network I/O system calls. Furthermore, no node needs to process information which is not relevant to termination at that point in time.

At this stage, we have presented an implementation of a language runtime for concurrent *and* distributed programming. We have repeatedly mentioned that *causality* is important for both the concurrent and the distributed setting. On many-core machines, causality is easy to achieve, because sending a message to an actor can be implemented using a single atomic operation [30]. Hence, causality in concurrent systems is a natural consequence of lock-free FIFO queues [30], because processors issue load and store instructions in order. However, guaranteeing causality in a distributed system is non-trivial, because messages may take arbitrary routes with varying latency to a destination. Due to performance concerns, we do want to avoid synchronizing clocks between runtime processes or to implement expensive vector clocks.

Inspired by why causality can be guaranteed with no overhead on many-core machines, we present a scheme that also guarantees causality in a distributed setting by construction. This scheme does not introduce "software overhead" of any kind, in a sense that we are not required to send messages, synchronize physical clocks or built partial event orderings within each runtime process by using vector clocks. However, we impose certain restrictions on the network topology. The following chapter provides a formal argument for message causality in networks that are based on a tree topology.

Chapter 4

Causality in Tree Network Topologies

Causal message delivery is a valuable property for concurrent and distributed programming. Furthermore, *Pony's* garbage collection scheme as well as the proposed distributed termination protocol *strongly* depend on causality. *Distributed Pony* enforces causality by the underlying tree network topology in combination with properties of TCP.

In the following, we give an informal view on the problem. A formal argument is provided to build the basis for a fully developed proof, which is subject to future work.

4.1 Informal View

In a distributed context, causal messaging requires to guarantee the property illustrated in Figure 4.1.

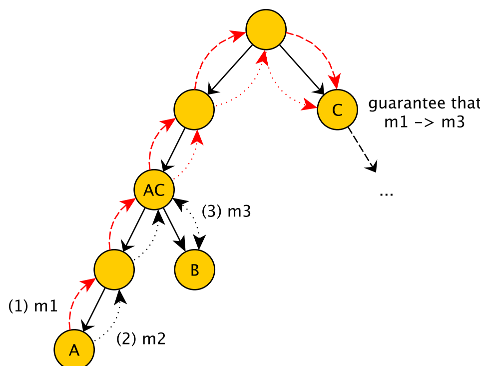


Figure 4.1: Causal Message Delivery in Tree Networks

If a `Node A` sends a message m_1 to another `Node C` *before* sending m_2 to `Node B`, which responds to the receipt of m_2 with a message m_3 to `Node C`, then `C` must receive m_1 *before* m_3. This property holds in the topology shown in Figure 4.1, because the path via which m_1 is send to `C` and the path via which m_3 is delivered to `C` share a common suffix (for this particular example). This is helpful, because the TCP protocol guarantees that messages written to the same channel arrive at the destination in the same order in which they were sent [22, 69, 71].

Hence, the important point is that m_1 arrives at the common ancestor of `A` and `B` (`AC`) *before* m_3. For the topology shown, this is clearly the case. `A` sends m_1 before m_2 and by TCP both messages arrive in that order at `AC`. Since m_3 is a cause of m_2, it must arrive at `AC` after m_2. The *happens before* relationship is transitive, in which case m_1 must have happened *before* m_3 at `AC`. From this point onwards both messages share the same path. TCP guarantees that m_1 is delivered to a process running on `Node C` *before* m_3. We can generalize this example to any constellation of `A`, `B` and `C` with an appropriate definition of "*paths*" within tree networks.

4.2 Formal Argument

An initial step towards a fully developed proof is to give an appropriate definition of *paths* and *routes* within tree network topologies. A path p can be formalized as a sequence of node identifiers, i.e.:

$$\text{Path} = (\text{NodeID})^* \tag{4.1}$$

where $p = p_1 \cdot n_1 \cdot n_2 \cdot p_3 \Rightarrow n_1, n_2$ are adjacent.

We define a *route* as a function that takes two node identifiers and returns the *shortest* path between the two nodes:

$$\text{route} : \text{Tree} \times \text{NodeID} \times \text{NodeID} \rightarrow \text{Path} \tag{4.2}$$

In a tree, a *shortest* path p_s does not form any "loops" (i.e. every node ID within p_s appears exactly once) and therefore *routes* are *unique*. Inspired by our initial thoughts given in example 4.1, we define a function $\circ$ that concatenates two routes and returns a path from the start of the first route to the destination of the second route:

$$\circ : route \times route \rightarrow path \tag{4.3}$$

Note that the resulting path may not necessarily be the shortest path. Hence, for any given tree T and node identifiers A, B and C:

$$\text{route}(T, A, C) \leq \text{route}(T, A, B) \circ \text{route}(T, B, C) \tag{4.4}$$

As illustrated in Figure 4.2, the idea is that the causal dependency between m_2 and m_3 can be modeled as a single message m_c which is routed from A to C via B. If $\text{route}(T, A, C) = \text{route}(T, A, B) \circ \text{route}(T, B, C)$, then causal message delivery between A, B and C is guaranteed by TCP. We are interested in the case, where a path from A to C is different from that of via B to C.

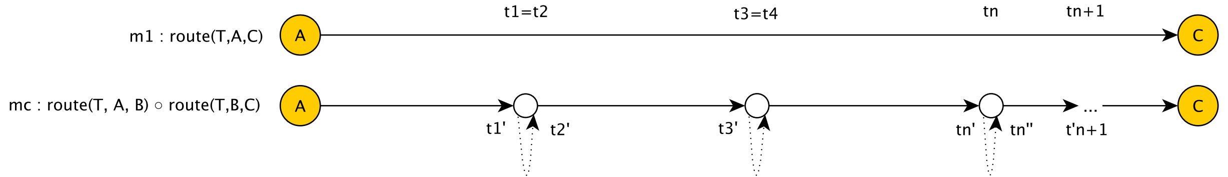


Figure 4.2: Routes from Source to Destination

A proof needs to show that m_1 precedes m_c at every point where m_c "returns" to the path shared with $\text{route}(T, A, C)$. Looking at Figure 4.2, we can establish the following relations:

- $t_1 < t'_1$, because the sending of m_1 *precedes* that of m_c and both messages traveled via a common prefix towards C. Furthermore, $t'_1 < t'_2$, because message delivery is not immediate. As a result, we can establish $t_1 < t'_1 < t'_2 < t'_3 < ...$
- $t_1 < t_3 \wedge t_3 < t'_3$

Hence, $t_n < t'_n$ and we can prove by induction over n that $t_{n+1} < t'_{n+1}$. Furthermore, we can establish that $t'_n < t''_n$. Combining these two properties with that of in-order delivery of TCP on common paths forms a convincing argument that causality is guaranteed in any kind of tree network topology. A fully developed proof is left for future work.

Chapter 5

Evaluation

Designing a benchmark suite for any kind of software application is enormously complex. Not only is it difficult to stress the interesting and critical parts of an application, but also to test an application based on realistic input data. For example, database management systems respond sensitively on the kind of queries to process and data value distribution they operate on [118]. This makes it difficult to reach a concluding result that can be used to predict an application's performance in a real-world scenario.

Another, probably more significant, challenge is *cognitive bias*. If a benchmark suite is developed by the same group of people as the system under test, it is likely that mostly those parts of an application that are expected (or even known) to perform well mainly influence the implementation of a corresponding benchmark. As a consequence, benchmark results would not be very representative. Thus, a benchmark must be fit for purpose (i.e. testing scenarios that are present in a productive environment) and at the same time as generic as possible. It is important to both benchmark the best case scenarios (those cases where the system is expected to perform well) as well as worst-case scenarios in a balanced manner. The knowledge gained from a benchmark should be that a system performs reasonably well in those cases for which it was built *and* that the worst-case scenarios do not cause a system to behave disastrously (quantified over some measurement). However, the development of *Pony* is still in a very early stage, because only an initial runtime system has been developed. Hence, we defer comparing *Pony* to other comparable programming languages to later stages. Besides being fundamentally error prone, developing a concise benchmark is time-consuming. For the purpose of this project *and* for the very first prototypes of *Concurrent* and *Distributed Pony* (where directions can still be changed) we are mostly interested understanding and evaluating the characteristics of the proposed mechanisms and algorithms. Hence, instead of developing our own generic benchmark suite, it is sensible to evaluate the system with simple test applications. We believe this is a good approach, because we can gain knowledge about how our implementation performs for a specific scenario.

For that reason, we have implemented a simple test application that allows us to test two different workload characteristics, *computation boundedness* and *message boundedness*. Compared to a single-node setting, we expect the runtime system of *Distributed Pony* to perform well for *computation bound* applications and to be slower for *message bound* applications. Before discussing the benchmark results of the runtime system, we give a brief overview of the test application, which we implemented from a specification available at [39]. Note that within this thesis, we evaluate the performance of the *distributed* part of the *Pony* runtime. However, we can implement a test program as for a concurrent setting and execute it (without any changes) on a network of computers.

Section 5.3 provides a showcase for computing the Mandelbrot set, illustrating the convenience of programming distributed applications based on the *Pony* runtime.

5.1 Test Application

We want our test application to produce two different kinds of workloads based on input parameters. On the one hand, we want to test the case where the dominant part of the application is the sending of messages between actors. On the other hand we want to put numerical workload on the available CPU resources but reduce the number of messages sent between actors to a minimum. Our test application is derived from a micro-benchmark published for another (concurrent) actor-based runtime called *libcppa* [39]. The source code is provided in Appendix B.

The test application creates a set of actors forming a ring. Each actor holds a pointer to its right-hand neighbor. The application accepts the following three input parameters:

- `--count <c>` determines the number of rings consisting of actors to be created
- `--size <s>` configures the number of actors in each ring
- `--pass <p>` sets an initial token value that is passed from actor to actor within each ring and decremented each time until it is zero. Once zero is reached, all actors within a ring terminate.

Each ring consists of $c - 1$ "chain-link" [39] actors and one master producing numerical workload, which is simulated using a brute-force approach for prime factorization of a large number. The main actor of the application creates `c`-many master (or factorization) actors and sends an initialization message to each of them. Upon receipt of such a message, a master enters its computation bound phase and factorizes a large number. This large number is a constant, therefore each master actor produces the same numerical workload [1]. Once it has completed the factorization, the master actor creates `s`-many actors to form a ring. This is the part where the application may enter its (potentially) message-bound phase depending on the given parameters. Figure 5.1 illustrates the micro-benchmark described above.

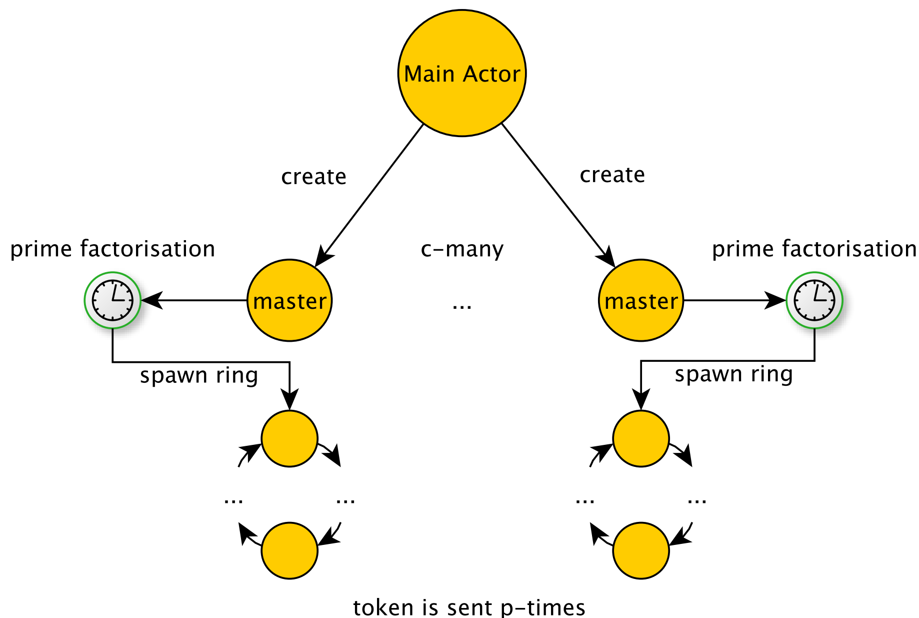


Figure 5.1: Micro-Benchmark

We expect *Distributed Pony* to perform well for a large number of rings (and therefore master actors) with a low value for passing the token. The network overhead of distributing the application should be substantial for a single ring and a high token value. For such a scenario, we expect *Distributed Pony* to be slower compared to the concurrent setting. The results are presented in the next section.

[1] The number is $28.350.160.440.309.881 = 86.028.157 * 329.545.133$

5.2 Message-bound vs. Computation-bound

The benchmarks have been carried out on a network consisting of three nodes, each equipped with the same hardware components:

- 1x Intel Core i7-2600 @ 3.40 GHz, Quad-Core (Hyper-Threaded) [70]
- 8 GB DDR3-1066 Main-Memory
- 1 Gbps Ethernet Connection

First, we discuss the performance results for the computation-bound scenario. Each run for a given set of parameters was executed 100 times. The execution time shown in the graphs is the average of these runs, based on the time of termination of the last node. Hence, the results presented include the latency of our proposed termination protocol.

Computation-bound Scenario

Our objective is to make the numerical workload the dominant part of the application. This can be achieved by choosing a large number of rings to be created, whilst keeping all other parameters to a minimum. In fact, the best case for the distributed setting should be when no application messages need to be sent over the network (except for the initialization messages). Hence, for the computation-bound benchmark, we set all message related parameters to zero and measured the execution time for a variable amount of factorization actors, i.e.:

```
--size 0 --count <var> --pass 0
```

Figure 5.2 and 5.3 show the performance results for up to 100 master actors. We have executed the benchmarks with and without utilizing logical cores (i.e hyper-threading).

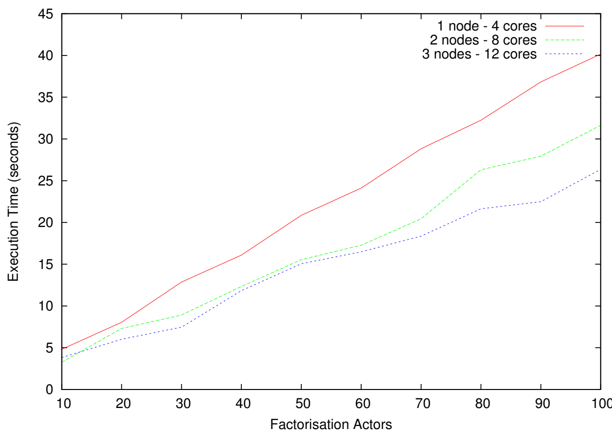


Figure 5.2: Using Physical Cores

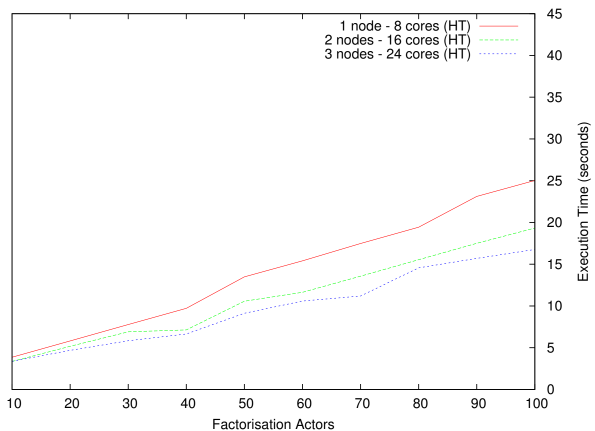


Figure 5.3: Using Logial Cores

The first observation is that, for this particular micro benchmark, *Distributed Pony* does not become slower with an increasing amount of available resources. The performance measurements for the single node configuration prove the characteristics of the test application. The additional execution time required with an increasing amount of factorization actors is (almost) linear, which speaks for the fact that the application is computation-bound, and not memory or network I/O bound.

The performance results show that the speedup achieved for a very small number of factorization actors is minimal. This is expected and is a logical consequence of our scheme for setting up a cluster of *Ponies*. Instead of having a static configuration and synchronizing the start of an application between participating peers, we start the test application on the master node and then connect new slave nodes. As a result, we do not benefit in the case of very small numbers of factorization actors, because all cores of the master node are occupied with executing numerical workload at

the beginning. Once the *Distribution Actor* of the master node is scheduled for the first time after slave nodes have been connected, most factorization actors have terminated. We can see this effect when comparing the two graphs. The benefit for 10 factorization actors is even smaller when using hyper threading. If we are unlucky and slave nodes establish their connection to the master node after it has started scheduling the prime factorization actors, 8 out of 10 computation-bound tasks are finished by the time the distributed work stealing scheduler has a chance to react for the first time. This is not a problem of the scheme for setting up a cluster of *Ponies* nor the scheduler, because most applications in the context of high performance computing and scalable applications are either long running or never terminate.

The application starts to benefit from distribution once a threshold of between 30 to 50 factorization actors is reached. This threshold is higher for the hyper-threaded environment, because the single node itself has more resources available, such that the amount of actors that can potentially be stolen is smaller from the beginning. In order to see whether the trend persists, we have executed the same benchmark with 1000 factorization actors. The results are shown in table 5.2.

Hyper-threading allows us to asses and reason about the performance of *Distributed Pony* more precisely, because we can compare the execution time of using 8 *logical* cores in a single-node environment with that of using 8 *physical* cores in a distributed environment. The speed up achieved using two nodes is about 50% of the benefit achieved through hyper-threading on a single machine (77.94s vs. 156.98s). Considering that running on two nodes includes network communication, deserialization and serialization, migration and the instantiation of remote actors (including heap, message queues etc.), the result seems satisfying. Furthermore, we have to keep in mind that these benchmarks have not been executed in a low-latency and isolated network.

The average scale factor achieved for three times the resources is about 1.5 for both the hyper-threaded and the physical core configuration. Note that the scale factor may vary greatly between benchmark runs for this particular test application, because the timing of when slave nodes establish their connection impacts how early they will be considered by the scheduler. For example, one run resulted in a scale factor of 2.19 for three times the resources. This fact leaves room for the assumption that *Distributed Pony* may perform excellent for real world scenario-type applications.

Table 5.1: Scale Factors - 100 factorization actors

Configuration	Physical	Hyper-Threading	Speedup Physical	Speedup Logical	combined
1 node	40.16s	25.03s	–	1.60	–
2 nodes	31.64s	19.33s	1.27	1.30	2.0
3 nodes	26.37s	16.76s	1.52	1.50	2.4

Table 5.2: Trend Analysis - 1000 factorization actors

Configuration	Physical	Hyper-Threading
1 node	399.84s	242.86s
2 nodes	321.90s	184.98s
3 nodes	276.37s	157.22s
Speed-up ($1 \rightarrow 3$)	1.44	1.54

Most importantly, we can observe a similar pattern in the increase of execution times for all configurations (with few exceptions, which may be due to statistical errors). This speaks for the efficiency of the scheduling and network implementation. Considering that *Distributed Pony* is a very first prototype, the performance results are encouraging. We expect that *Distributed Pony* also performs well on extremely large computer clusters due to the constant space complexity scheduling algorithm we have developed for this project and the optimality property of work-stealing. Future

work should attempt to execute benchmarks on a large cluster or super computer to verify our expectations.

We do not want to give the impression that distributing an application always ends up in performance benefits. In fact, a certain class of actor-based applications does not perform well in a distributed context. If sending messages is the dominant part of an application, distributed computing may not be a winning strategy. The network overhead involved for sending messages between nodes causes that message-bound applications involve higher latency compared to a single-node environment. We examine this scenario in the following.

Message-bound Scenario

We can make our test application message-bound by adopting the inverse approach to the one used for the previous benchmarks. However, in order to achieve *true* message-boundedness we need to be careful in choosing the right parameters and consider various characteristics of our implemented runtime system. Remember that *Distributed Pony* does not provide a user-level scheduler (and we do not want it to). Hence, we have no control over which actors are migrated to which node and when.

However, it is also not enough to just spawn a single ring. Due to the token-ring type of message passing, there is always only one actor of each ring at a time that is unblocked. Therefore, our distributed scheduler *might* never migrate an actor if there were only a single ring. Consequently, our test application is guaranteed to be message-bound if we spawn *more* rings as there are local cores available and provide it with a high token value.

Note that the size of each ring is important. Large rings would cause fewer messages to be sent over the network. Since only one actor of each ring is unblocked at a time (and therefore on any of the work stealing queues), the scheduler might end up migrating only a single actor of a ring to another node. Since the token value is decremented each time, most of the message workload would remain local (depending on the token value). Consequently, we want the size of the ring to be exactly 2, such that network communication must happen each time the token is passed on to the next neighbor. Message-boundedness is difficult to test in combination with a distributed work stealing scheduler, because those applications may have many actors, but mostly no *work* to steal. Note that it is more difficult to compare the results based on an increasing amount of available resources, because we need to spawn more rings depending on the amount of cores available in the entire system:

```
--size 2 --count <total_core_count> --pass <var>
```

As previously, we measure the average execution time of 100 runs. The results are shown in Figure 5.5. Since for this particular scenario and application more nodes do not deliver any more insights, we only execute this benchmark on two nodes and do not utilize logical cores.

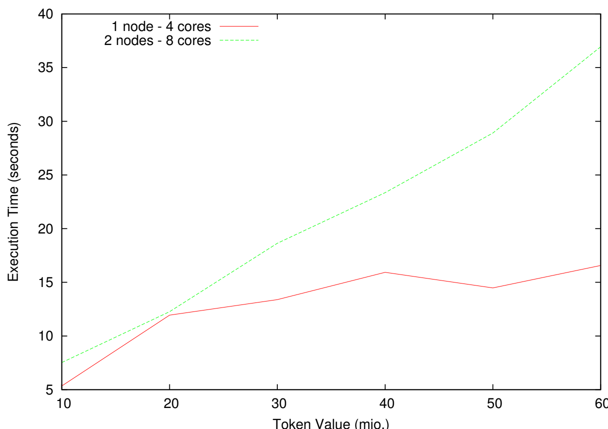


Figure 5.4: Message-Bound Test Scenario

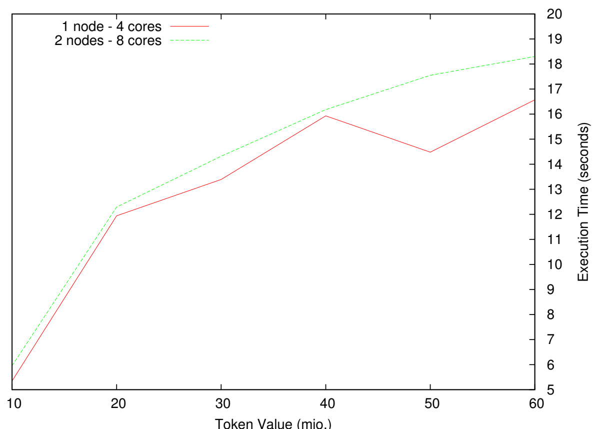


Figure 5.5: Constant number of rings

There is evidence to see that network communication introduces a substantial overhead. The results may look disastrous, but we also have to consider that in order to stimulate the work stealing scheduler, for this particular application, we are required to send double the amount of application messages, caused by spawning double the amount of rings. Furthermore, each migrated actor causes two delegation messages to be sent for one application message. The important point is that this discussion emphasizes that work stealing is about stealing work and not actors. Hence, at least to some extent, a scheduler based on work stealing avoids unnecessary migration for message-bound applications by construction. We can see this effect in Figure 5.4, where we have executed the benchmark on two nodes with the same amount of rings as compared to the concurrent setting. There is basically no difference between the two, because the work stealing scheduler does not migrate many actors. Even though using as many rings as cores on one machine are available, there might still be some actors migrated, because the *Distribution Actor* may steal an unblocked actor that is on the same scheduling queue as the *Distribution Actor* itself.

The analysis of the message-bound performance results is vague, because contrary to the previous benchmark, the key to interpret the results correctly is to know which actors haven been migrated. This information is not available. For the computation-bound scenario this was not important, because there were only factorization actors to be migrated. Furthermore, factorization actors did not send messages. Consequently there is no dependency between actors that might influence the performance results.

The discussion on the subtleties of the behavior of the test application and the work stealing scheduler of *Distributed Pony* shows that it might not be enough to steal work on a random basis. Instead, it might be necessary to look at the communication patterns of actor-based applications. This topic is subject for future work. We believe that this may yield an interesting discussion on programming best-practices for the actor programming model.

5.3 Programming Example - Mandelbrot Fractals

Evaluating an implementation is not only about discussing its performance. In the context of programming languages and runtimes, we also have to consider the benefits it delivers to the programmer. Since the *Pony* syntax, type system and compiler is not fully developed yet, we provide a simple example of how an actor-based concurrent and distributed application can be programmed in *C* using the runtime extended for this project. The example shall be to compute the *Mandelbrot set* on arbitrarily many cores and nodes.

We have previously discussed in chapter 2, that each *Pony*-actor is given a type, a set of message conversion functions, a top-level trace function as well as a message dispatch handler. For simplicity and because this is just a show case, we decide to spawn an actor for each pixel of a 400×400 Mandelbrot image. This is not optimal from a performance point of view, because the resulting application is message-bound and causes many actors to be created. However, we would be required to provide a top-level trace function for variable length byte-arrays if we wanted each actor to process a chunk of the overall image.

Each application requires a *main actor*, which is sent a `PONY_MAIN` message by the runtime system including any command line arguments in order to start the application. The main actor of our sample application is responsible for managing a 400×400 matrix of color values and spawns an actor for each element. We provide three types of application messages, as defined in listing 5.1:

- `MSG_COMPUTE_PIXEL` is sent to each spawned worker actor. The arguments are a reference to the main actor and the coordinates of the pixel for which the color value should be computed.

- `MSG_COLLECT_RESULT` is the response of each worker to the main actor containing the computed color value as well as the pixel coordinates for that value. We need to provide the coordinates, because worker actors may respond in any order.

  We limit this application to a resulting Mandelbrot image that is black and white. Hence, a single 8-bit integer is sufficient for the result.

- `MSG_FINISH` is sent to the main actor by itself once all worker actors have responded. The resulting matrix is written to `stdout`.

Our test application could be extended to allow for arbitrarily large Mandelbrot images to be computed and to be able to adjust the resolution and zoom factor. For the purpose of this example, we disregard such features.

Listing 5.1: Message Types for Mandelbrot Application

```
1 static message_type_t m_compute_pixel = {3, {NULL}, {0}, {PONY_ACTOR,
      PONY_PRIMITIVE16, PONY_PRIMITIVE16}};
2
3 static message_type_t m_collect_result = {3, {NULL}, {0}, {PONY_PRIMITIVE16,
      PONY_PRIMITIVE16, PONY_PRIMITIVE8}};
4
5 static message_type_t m_finish = {0, {NULL}, {0}, {PONY_NONE}};
6
7 static message_type_t* message_type(uint64_t id)
8 {
9   switch(id)
10   {
11     case MSG_COMPUTE_PIXEL: return &m_compute_pixel;
12     case MSG_COLLECT_RESULT: return &m_collect_result;
13     case MSG_FINISH: return &m_finish;
14   }
15   return NULL;
16 }
```

Worker actors do not need to maintain any data and therefore we are not required to provide a top level trace function, because there is no data to serialize or deserialize for migration. The type descriptor of a worker actor is provided in listing 5.2.

Listing 5.2: Type descriptor for Mandelbrot actor

```
1 static actor_type_t worker =
2 {
3   0,
4   NULL,
5   message_type, /* message type conversion function from listing 5.1 */
6   dispatch      /* pointer to message dispatch handler */
7 };
```

The message handler for `PONY_MAIN` creates 400×400 workers and sends a message to start computing a pixel value to each of those actors. The runtime function `pony_sendv` allows to send a vector of arguments in a message:

Listing 5.3: Spawning actors and sending a vector of arguments

```
1 arg_t args[3];
2
3 for(uint16_t j = 0; j < 400; j++)
4 {
5   for(uint16_t i = 0; i < 400; i++)
6   {
7     args[0].p = this;
8     args[1].i = j;
9     args[2].i = i;
10
11     pony_sendv(pony_create(&worker), MSG_COMPUTE_PIXEL, 3, args);
12   }
13 }
```

A given pixel coordinate c (expressed as complex number) belongs to the Mandelbrot set, if the absolute value of z_n is bounded for $n \to \infty$ [94]:

$$z_{n+1} = z_n^2 + c \tag{5.1}$$

Hence, the computed result is an approximation and we exit if the number of iterations exceeds a certain threshold (in this example $2^8 - 1$) or if z_n is determined not to be part of the Mandelbrot set (i.e. if either of its real or imaginary part is larger than 2). This is also known as the *escape time* algorithm. The value that each worker reports to the main actor is the number of executed iterations. Note that the equation above can be implemented efficiently with C99's implementation of complex numbers. The interested reader is referred to [47] for additional information. Listing 5.4 shows the source code for computing the color value of a pixel at position (x, y).

Listing 5.4: Compute color value

```
1 case MSG_COMPUTE_PIXEL:
2 {
3   actor_t* collector = (actor_t*)argv[0].p;
4   uint16_t x = (uint16_t)argv[1].i;
5   uint16_t y = (uint16_t)argv[2].i;
6
7   double complex c, z;
8   uint8_t n;
9
10  /* center the image and zoom out */
11  c = -0.3+(x-400/2)*0.007 + I * ((y-400/2)*0.007);
12  z = 0.0;
13  n = 0;
14
15  while((cabs(z) < 2.0) && (n != 255))
16  {
17    z = z*z + c;
18    n++;
19  }
20
21  arg_t res[3];
22  res[0].i = x;
23  res[1].i = y;
24  res[2].i = n;
25
26  pony_sendv(collector, MSG_COLLECT_RESULT, 3, res);
27
28  break;
29 }
```

As for any other C application, we need to provide a *main* function that starts the runtime in order to invoke the main actor. For this purpose, *Pony* provides a function called `pony_start`, as shown in listing 5.5. Since the main actor executes I/O operations by writing the resulting color values to `stdout`, we need to avoid that the scheduler migrates this actor to another node. This can be achieved by using the function `pony_pin_actor`.

In future versions of *Pony*, pinning an actor explicitly is not required, because we can determine whether an actor implementation has dependencies to I/O related system calls.

Listing 5.5: A Pony program's main function

```
1 int main(int argc, char** argv)
2 {
3   actor_t* main_actor = pony_create(&worker);
4   pony_pin_actor(main_actor);
5
6   pony_start(argc, argv, main_actor);
7 }
```

The application can be executed in a concurrent setting just like any other *C* application. Computing the Mandelbrot set on a cluster of *Ponies* requires to start the application on a master node with the following parameters, without the need for changing the code or recompilation:

```
./mandelbrot --ponydistrib --ponymaster > output.data
```

Slave nodes can be connected at any point in time (before the master terminates) like the following:

```
./mandelbrot --ponydistrib --ponyconnect <host_master> <port>
```

Runtime processes on slave nodes do not spawn a main actor, instead they just stay idle until their parent node migrates actors to execute. The image produced is shown in Figure 5.6. Note that because the application returns a file of bytes representing color values, we need to convert the result to `jpeg` format or similar. On UNIX based systems, this can be achieved using a program called "convert" [92], which is a member of the *ImageMagick*-suite tools [93]:

```
convert -depth 8 -size 400x400 gray:output.data output.jpg
```

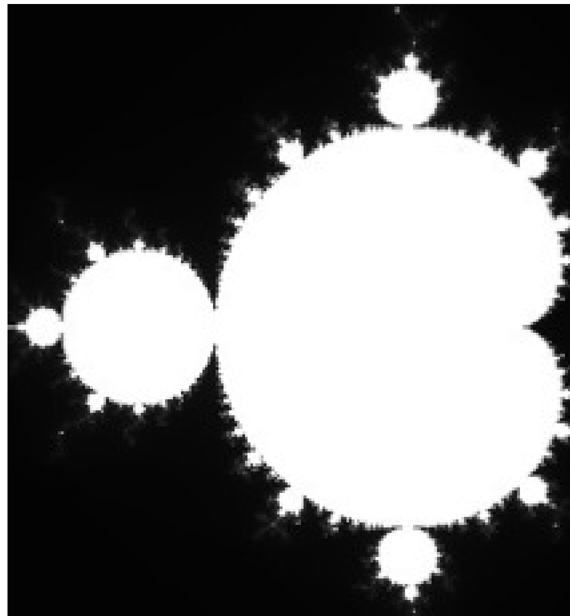


Figure 5.6: Back-and-White colored cut-out of Mandelbrot-Set

Chapter 6

Conclusion

6.1 Contributions

This thesis builds on the observation that the Actor programming model is inherently well suited for distributed computing [2]. We proposed an implementation of a programming language runtime called *Pony* that seamlessly binds the two aspects of concurrent and distributed computing, without exposing the programmer to anything related to the underlying resource topology. Any program written based on our runtime can be executed in a concurrent setting or on arbitrarily large computer clusters without changing the code or recompilation.

In the following we summarize the contributions of this work. The first item is asynchronous I/O multiplexing, which required low-level socket programming and consideration of best practices. The remaining items required the development of novel algorithms and protocols.

Asynchronous Network I/O Multiplexing

We have implemented an efficient network layer for asynchronous I/O multiplexing based on *poll*. Messages between nodes are dispatched efficiently using a framed message protocol without expensive delimiter parsing. Serialization and deserialization of any structure and message is supported through a trace mechanism in combination with a stream buffer data type. Any message, independent of its size, is written to the network with as least system calls as possible. I/O operations are implemented in a non-blocking fashion, which avoids unnecessary waiting for messages.

Any network communication, such as remote actor application messages, is implemented by a *Distribution Actor*, which is responsible for any tasks related to distribution. The *Distribution Actor* is scheduled by repeatedly sending a message to it. Busy waiting is avoided using a zero-valued nano sleep after each invocation of the *Distribution Actor*.

Joining Nodes to a Cluster of Ponies (section 3.3.2)

A cluster of *Ponies* is based on a k-Tree network topology, is easy to set up and nodes can be added at runtime without any reconfiguration. We proposed a joining algorithm for adding slave nodes to an existing cluster that guarantees that the tree network is *almost* balanced at any point in time. This means that the imbalance factor is either ± 1 or the tree is perfectly balanced and therefore a *complete* k-Tree.

The same algorithm is used to route messages from source to destination within a *Pony* network. The next hop towards a destination can be calculated efficiently if the maximum number of children per node is a power of 2.

Distributed Work Stealing Scheduler (section 3.5.4)

The centerpiece of the runtime developed in the context of this thesis is a distributed work stealing scheduler, which allows to live-migrate actors at runtime. We developed a hierarchical scheduling algorithm specifically tailored for tree network topologies with constant space complexity, indepen-

dent of the size of a *Pony* cluster. Our approach is based on a *push* and *pull* scheme, where nodes report the free core count of themselves and their children transitively to their parent node. We prioritize sending work down the tree over sending work up the tree. This decision is based on a statistical argument, namely that the path to reach a node which is not used to capacity is longer via a parent node than via a child node. Furthermore, the proposed algorithm guarantees that busy intermediate nodes do not cause nodes further down the tree to suffer from starvation.

Work stealing is provably optimal (within some constant factor) for task and data parallelism. Note that a distributed work stealing scheduler for actor-based applications is not about stealing *actors*, but about stealing *work* to be executed by actors. Hence, our algorithm does not attempt to distribute all actors in a system uniformly among a set of nodes in a distributed setting. The scheduler is optimized for throughput, which in the long term also optimizes for latency.

Causal Message Delivery in Distributed Systems (chapter 4)

Pony guarantees *causal message delivery* for both the concurrent *and* the distributed setting without any additional software overhead. In a single-node configuration, causality is a natural consequence of atomic memory operations [30] used to synchronize adding messages to an actors mailbox.

Instead of using logical or vector clocks to impose a partial order on a set of events, we enforce causality in a distributed system through a tree network topology in combination with characteristics of the TCP protocol. A formal argument is provided in section 4.2.

We believe that causal messaging is a valuable property for programming actor-based concurrent and distributed applications. A good example for this is the *Pony* runtime itself, where efficient and fully concurrent garbage collection of actors as well as distributed termination is possible because of causality. Not only is it a matter of efficiency, but also important for how we reason about concurrent and distributed programs.

Distributed Garbage Collection of Actors (section 3.7)

The concurrent version of *Pony*, developed prior to this project, implements fully concurrent garbage collection of actors [30]. In this thesis, we proposed an extension to this scheme for the distributed context, optimized for tree network topologies.

Reference counting alone is not sufficient. Actors may reference other actors and therefore produce cyclic garbage, which is uncollectable using reference counting alone. In a concurrent setting, *Pony* employs an actor as cycle detector to cater for cyclic garbage. Our extended scheme for distributed systems provides a cycle detector on every node of a system, and therefore allows to collect local cyclic garbage with no overhead compared to the concurrent setting. If a cycle consists of actors located on different nodes, our algorithm guarantees that the decision to collect such a cycle can be made at the earliest common ancestor of the nodes holding actors that participate in that cycle. Thus, we avoid unnecessary sending of information up the tree. Causality allows us to avoid a complicated communication protocol between all nodes we need to consider for collecting a cycle.

Distributed Termination (section 3.8)

A consequence of garbage collection for actors is that a *Pony* programmer is not required to maintain the lifetime of an actor. We proposed a protocol for fully automatic termination of distributed and actor-based applications. The causality property of *Pony* in combination with a conf-ack protocol allows us to implement a central *termination actor* that detects *distributed quiescence* and coordinates the termination of all nodes participating in a cluster of *Ponies*.

Benchmarks (chapter 5)

We have implemented a micro-benchmark to give an *initial* evaluation of our implementation. The results show that we achieve considerable speedup for computation-bound scenarios. We discussed that scaling actor-based applications on a distributed network of computers is largely dependent on the communication pattern of a particular application. If the sending of messages is the dominant part of an application, distribution may not be a winning strategy.

6.2 Future Work

The implemented runtime is a first step towards a complete programming language. Furthermore, the wide scope of the topics discussed within this thesis leaves room for many directions of future continuation of this research.

Improvement of I/O Multiplexing

Scheduling the *Distribution Actor* by sending a message to it after each invocation is problematic for computation-bound scenarios, where the *Distribution Actor* is not busy most of the time. The current approach generates unnecessary polling for I/O events and schedules the *Distribution Actor* independently of the utilization level of the node it is running on. Although we cannot entirely avoid to probe for I/O without reading data from network sockets, it may be sensible to introduce one layer of indirection. We can imagine to add another asynchronous mechanism as kernel timer to send a probe message to the *Distribution Actor*.

The interval within which probing for I/O is scheduled could be set dynamically based on application characteristics and CPU utilization. For example, if every core on a machine is used to its capacity we could increase the interval of I/O scheduling. If an application is message-bound, the interval may be shortened.

Furthermore, future work should provide a discussion on whether a single *Distribution Actor* might become a bottleneck. Note that this discussion is not important because of message-bound applications, which might not scale well in a distributed setting anyway, but because of applications that yield large amounts of actors. More extensive benchmarking may evaluate whether a single *Distribution Actor* might involve unacceptably high latency for migration for these type of applications.

Failure Detection and Dynamic Tree Topology

We have not addressed the problem of partial failure in distributed systems. The question is, whether failure can be handled in a transparent manner, similar to the scheduling of actors presented in this work, such that the programmer is not exposed to any of these issues.

A possible idea could be to introduce a *Failure Detector-Actor* on every node. A node being suspected could cause a leader election [50, 75] to be triggered between the children of the crashed node. The elected leader could be used as the new parent node. Implementing such a feature is difficult, because we dynamically change the topology of the network but at the same time need to maintain the causality property of *Pony*.

If nodes can fail, we need to ensure that messages are handled appropriately if a destination becomes suspected. Note that not only a destination might have crashed, but potentially any intermediate node on a path from source to destination. This might require a cache for outgoing remote messages and a *conf-ack* protocol to ensure message delivery.

We consider failure detection to be one of the most important challenges to be addressed by future versions of *Pony*.

Locality-aware Tree Network Topology

The proposed joining algorithm can be improved in various ways. Instead of solely deciding the location of a new slave node based on its ordinal number, we could also take the network latency

between nodes into account. In order to improve data locality of *Pony's* hierarchical work stealing scheduler, we could determine the delegation path of a new slave node based on which node can ping the new slave node fastest (and then decide its node ID), whilst guaranteeing that the tree is *almost* balanced. Of course, locality and balancing the tree may be conflicting requirements. Hence, it may be sensible to provide a mechanism that enables the user to prioritize properties of the underlying tree topology. Also, it might be desirable to allow for dynamic re-arrangement of the tree. For example, the number of children per node might be changed at runtime for reducing latency. Again, the main challenge of a dynamic topology is maintaining causality.

Work Stealing and Actor Migration

The proposed hierarchical work stealing mechanism in combination with live-migration of actors requires fine tuning. It might be advantageous to migrate actors on purpose, rather than only for work stealing. This may be sensible for remote messages with a size that is above a certain threshold. Instead of transferring the message, it might be more efficient to migrate the receiving actor to the same node as the sender.

Our *proxy* mechanism could be extended for this purpose. Instead of delegating large messages, a proxy could signal the *Distribution Actor* to migrate the recipient (which might conflict with other concurrent migration requests). Provided we can maintain causality, the proxy itself could hold on to the message until the true actor arrives. Eventually, the proxy becomes a local true actor and processes the large message.

Work stealing on its own may not be sufficient. It might be sensible to also consider the communication patterns of an application in order to avoid work stealing the message-bound part of an application. We believe that a discussion on communication patterns of actor-based applications could lead to interesting insights for programming best-practices for the actor programming model.

Compiler, Type System and Formal Models

The development of the *Pony* runtime has advanced to a stage at which we should start to develop a compiler for the *Pony* language. Furthermore, especially in the context of message-passing systems, the development of a type system is of great importance and has a large impact on the efficiency of a runtime system.

Most importantly, formal models should be provided for the mechanisms proposed in this thesis to allow for soundness proofs. In later stages, this also applies for developing the operational semantics of the *Pony* language.

Generalization

Throughout the time of this project, we have experienced that some solutions proposed were applicable to various problems we faced, such as the acknowledgment protocol for cycle detection. Consequently, future research should investigate to which extent the work carried out in this project may go beyond the context of *Pony* in order to solve problems in a distributed context.

Publications

This thesis provides material for several publications:

- A *Pony* Paper
- Hierarchical Garbage Collection of Actors and Objects in Distributed Systems
- Causality in Distributed Systems by Construction
- Hierarchical Work Stealing of Actors in Distributed Systems

Appendix A

Non-blocking reads

A.1 Pseudocode - socket read handler: collect()

Listing A.1: Pseudocode of collect() - handling EOF is omitted

```
1 static actor_t* collect(message_buf* msg_buf, const int sockfd)
2 {
3   ssize_t ret;
4
5   if(!msg_buf->collect_started)
6     msg_buf_prepare();
7   do
8   {
9     ret = read(sockfd, msg_buf->offset, msg_buf->bytes_left);
10
11    if(ret > 0)
12    {
13      msg_buf->offset += ret;
14      msg_buf->bytes_left -= ret;
15    }
16  } while(errno != EWOULDBLOCK && msg_buf->bytes_left > 0);
17
18  give_up_if_blocked();
19
20  if(msg_buf->header_complete == false)
21    check_and_prepare_for_body();
22
23  /* read message body */
24   if(msg_buf->bytes_left > 0)
25     return collect(msg_buf, sockfd);
26
27   set_handler();
28   return receiving_actor();
29 }
30
31 static void handle_read(message_buf* msg_buf, const int sockfd)
32 {
33   actor_t* recv = 0;
34
35   if((recv = collect(msg_buf, sockfd)) != 0)
36     msg_buf->dispatch(recv, msg_buf->msg_body);
37 }
```

Appendix B

Micro Benchmark

Listing B.1: Micro Benchmark to simulate workloads

```
1 #define __STDC_FORMAT_MACROS
2 #include <pony/pony.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <string.h>
6 #include <inttypes.h>
7 #include <assert.h>
8
9 typedef struct ring_t
10 {
11   actor_t* worker;
12   actor_t* next;
13   int size;
14   int pass;
15   int repeat;
16 } ring_t;
17
18 enum
19 {
20   MSG_INIT,
21   MSG_NEXT,
22   MSG_PASS,
23   MSG_WORK
24 };
25
26 static void trace(void* p);
27 static message_type_t* message_type(uint64_t id);
28 static void dispatch(actor_t* this, void* p, uint64_t id, int argc, arg_t* argv);
29
30 static actor_type_t type =
31 {
32   sizeof(ring_t),
33   trace,
34   message_type,
35   dispatch
36 };
37
38 static message_type_t m_init = {3, {NULL}, {0}, {PONY_PRIMITIVE64, PONY_PRIMITIVE64
     , PONY_PRIMITIVE64}};
39 static message_type_t m_next = {1, {NULL}, {0}, {PONY_ACTOR}};
40 static message_type_t m_pass = {1, {NULL}, {0}, {PONY_PRIMITIVE64}};
41 static message_type_t m_work = {0, {NULL}, {0}, {PONY_NONE}};
42
43 static void trace(void* p)
44 {
45   ring_t* d = p;
46   pony_traceactor(&d->worker);
47   pony_traceactor(&d->next);
```

```
48   pony_trace32(&d->size);
49   pony_trace32(&d->pass);
50   pony_trace32(&d->repeat);
51 }
52
53 static message_type_t* message_type(uint64_t id)
54 {
55   switch(id)
56   {
57     case MSG_INIT: return &m_init;
58     case MSG_NEXT: return &m_next;
59     case MSG_PASS: return &m_pass;
60     case MSG_WORK: return &m_work;
61   }
62
63   return NULL;
64 }
65
66 static uint64_t factorize(uint64_t n, uint64_t count, uint64_t* list)
67 {
68   if(n <= 3)
69   {
70     list[0] = n;
71     return 1;
72   }
73
74   uint64_t d = 2;
75   uint64_t i = 0;
76
77   while(d < n)
78   {
79     if((n % d) == 0)
80     {
81       list[i++] = d;
82       n /= d;
83     } else {
84       d = (d == 2) ? 3 : (d + 2);
85     }
86   }
87
88   list[i++] = d;
89   return i;
90 }
91
92 static void test_factorize()
93 {
94   uint64_t list[2];
95   uint64_t count = factorize(86028157UL * 329545133UL, 2, list);
96
97   if((count != 2) || (list[0] != 86028157) || (list[1] != 329545133))
98   {
99     printf("factorization error");
100   }
101 }
102
103 static actor_t* spawn_ring(actor_t* first, int size, int pass)
104 {
105   actor_t* next = first;
106
107   for(int i = 0; i < (size - 1); i++)
108   {
109     actor_t* actor = pony_create(&type);
110     pony_sendp(actor, MSG_NEXT, next);
111     next = actor;
112   }
113
```

```
114   if(pass > 0) pony_sendi(first, MSG_PASS, pass * size);
115   return next;
116 }
117
118 static void dispatch(actor_t* this, void* p, uint64_t id, int argc, arg_t* argv)
119 {
120   ring_t* d = p;
121
122   switch(id)
123   {
124     case PONY_MAIN:
125     {
126       int margc = argv[0].i;
127       char** margv = argv[1].p;
128       int size = 50;
129       int count = 20;
130       int pass = 10000;
131       int repeat = 5;
132
133       for(int i = 1; i < margc; i++)
134       {
135         if(!strcmp(margv[i], "--size"))
136         {
137           if(margc <= (i + 1))
138           {
139             return;
140           }
141
142           size = atoi(margv[++i]);
143         } else if(!strcmp(margv[i], "--count")) {
144           if(margc <= (i + 1))
145           {
146             return;
147           }
148
149           count = atoi(margv[++i]);
150         } else if(!strcmp(margv[i], "--pass")) {
151           if(margc <= (i + 1))
152           {
153             return;
154           }
155
156           pass = atoi(margv[++i]);
157         } else if(!strcmp(margv[i], "--repeat")) {
158           if(margc <= (i + 1))
159           {
160             return;
161           }
162
163           repeat = atoi(margv[++i]);
164         } else {
165           return;
166         }
167       }
168
169       argv[0].i = size;
170       argv[1].i = pass;
171       argv[2].i = repeat;
172
173       for(int i = 0; i < count; i++)
174       {
175         pony_sendv(pony_create(&type), MSG_INIT, 3, argv);
176       }
177       break;
178     }
179
```

```
180     case MSG_INIT:
181     {
182       d = pony_alloc(sizeof(ring_t));
183       pony_set(d);
184
185       d->worker = pony_create(&type);
186       d->size = argv[0].i;
187       d->pass = argv[1].i;
188       d->repeat = argv[2].i;
189
190       pony_send(d->worker, MSG_WORK);
191       d->next = spawn_ring(this, d->size, d->pass);
192       break;
193     }
194
195     case MSG_NEXT:
196     {
197       d = pony_alloc(sizeof(ring_t));
198       pony_set(d);
199
200       d->worker = NULL;
201       d->next = argv[0].p;
202       d->size = 0;
203       d->pass = 0;
204       d->repeat = 0;
205       break;
206     }
207
208     case MSG_PASS:
209     {
210       if(argv[0].i > 0)
211       {
212         pony_sendi(d->next, MSG_PASS, argv[0].i - 1);
213       } else {
214         assert(d->repeat > 0);
215         assert(d->worker != NULL);
216         d->repeat--;
217
218         if(d->repeat > 0)
219         {
220           pony_send(d->worker, MSG_WORK);
221           d->next = spawn_ring(this, d->size, d->pass);
222         }
223       }
224       break;
225     }
226
227     case MSG_WORK:
228     {
229       test_factorize();
230       break;
231     }
232   }
233 }
234
235 int main(int argc, char** argv)
236 {
237   actor_types[0] = &type;
238
239   return pony_start(argc, argv, pony_create(&type));
240 }
```

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM.

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

[3] Apache. Thrift. `http://thrift.apache.org`. Accessed: 27/07/2013.

[4] Apple. thread_policy_set(). `http://developer.apple.com/library/mac/#releasenotes/Performance/RN-AffinityAPI/index.html`. Accessed: 31/07/2013.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[6] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[7] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for unix. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 19–19, Berkeley, CA, USA, 1999. USENIX Association.

[8] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.

[9] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. Technical report, Cornell University, Ithaca, NY, USA, 1987.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[11] Ben A. Blake. Assignment of independent tasks to minimize completion time. *Software, Practice and Experience*, 22(9):723–734, 1992.

[12] Wolfgang Blochinger and Wolfgang Kchlin. The design of an api for strict multithreading in c++. Lecture Notes in Computer Science, pages 722–731. Springer.

[13] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the perfect benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, 1992.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[15] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[16] Jonas Bonér. Akka - Building Concurrent and Distributed Applications based on the JVM. `http://www.akka.io`. Accessed: 3/09/2013.

[17] Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. ACM.

[18] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[19] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, March 1995.

[20] Peter A. Buhr and Ashif S. Harji. Implicit-signal monitors. *ACM Trans. Program. Lang. Syst.*, 27(6):1270–1343, November 2005.

[21] Thomas L. Casavant, Jon, and G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14:141–154, 1988.

[22] Vinton G. Cerf and Robert E. Icahn. A protocol for packet network intercommunication. *SIGCOMM Comput. Commun. Rev.*, 35(2):71–82, April 2005.

[23] Tushar D Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. Technical report, Ithaca, NY, USA, 1994.

[24] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[25] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[26] Dominik Charousset. libcppa - An implementation of the Actor Model for C++. `http://libcppa.blogspot.co.uk`. Accessed: 3/09/2013.

[27] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[28] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[29] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[30] Sylvan Clebsch and Sophia Drossopoulou. Fully Concurrency Garbage Collection of Actors on Many-Core Machines. To appear, OOPSLA, 2013.

[31] Mozart Consortium. The Mozart Programming Language. `http://http://www.mozart-oz.org`. Accessed: 11/07/2013.

[32] Microsoft Corp. Windows Asynchronous Procedure Calls. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx`. Accessed: 29/07/2013.

[33] Microsoft Corp. Windows I/O Completion Ports. http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx. Accessed: 29/07/2013.

[34] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[35] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, August 2007.

[36] Peter Deutsch. The Eight Fallacies of Distributed Computing. https://blogs.oracle.com/jag/resource/Fallacies.html. Accessed: 10/07/2013.

[37] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[38] Erricson. The Erlang Programming Language. http://www.erlang.org. Accessed: 11/07/2013.

[39] Dominik Charousset et. al. libcppa vs. Erlang vs. Scala Performance (Mixed Scenario). http://libcppa.blogspot.co.uk/2012/02/libcppa-vs-erlang-vs-scala-performance.html. Accessed: 1/09/2013.

[40] Martin Odersky et al. The Scala Programming Language. http://www.scala-lang.org. Accessed: 11/07/2013.

[41] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[42] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '83, pages 1–7, New York, NY, USA, 1983. ACM.

[43] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.

[44] The Free Software Foundation. BSD Socket API - socket(). http://man7.org/linux/man-pages/man2/socket.2.html. Accessed: 5/08/2013.

[45] Inc. Free Software Foundation. The GNU Standard C Library. http://www.gnu.org/software/libc/manual/html_node/Processor-Resources.html. Accessed: 31/07/2013.

[46] Inc. Free Software Foundation. Thread-local variables. http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Thread_002dLocal.html. Accessed: 7/08/2013.

[47] The Free Software Foundation. C99 standard - complex.h. http://www.gnu.org/software/libc/manual/html_node/Complex-Numbers.html#Complex-Numbers. Accessed: 5/09/2013.

[48] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[49] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[50] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[51] GCC GNU and Intel. Atomic builtins - __sync_bool_compare_and_swap. `http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Atomic-Builtins.html#Atomic-Builtins`. Accessed: 29/07/2013.

[52] GCC GNU and Intel. Atomic builtins - __sync_lock_test_and_set. `http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Atomic-Builtins.html#Atomic-Builtins`. Accessed: 29/07/2013.

[53] A. Ghafoor and I. Ahmad. An efficient model of dynamic task scheduling for distributed systems. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 442–447, 1990.

[54] The Object Management Group. The CORBA Standard. `http://www.corba.org`. Accessed: 10/07/2013.

[55] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM J. Res. Dev.*, 27(6):530–544, November 1983.

[56] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '91, pages 120–132, New York, NY, USA, 1991. ACM.

[57] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, July 1990.

[58] P.B. Hansen. The programming language concurrent pascal. *Software Engineering, IEEE Transactions on*, SE-1(2):199 –207, june 1975.

[59] Per Brinch Hansen. Monitors and concurrent pascal: a personal history. *SIGPLAN Not.*, 28(3):1–35, March 1993.

[60] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. Technical report, Mountain View, CA, USA, 2005.

[61] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM.

[62] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999.

[63] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[64] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[65] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[66] The IEEE and The Open Group. POSIX Threads - IEEE Std 1003.1, The Open Group Base Specifications Issue 6. http://pubs.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html. Accessed: 31/07/2013.

[67] Oracle Inc. The Java Programming Language. http://www.java.com. Accessed: 10/08/2013.

[68] Oracle Inc. Java RMI. http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html. Accessed: 10/07/2013.

[69] Information Sciences Institute University of Southern California. Transmission Control Protocol. http://tools.ietf.org/html/rfc793. Accessed: 2/07/2013.

[70] Intel. Core i7-2600 Product Page. http://ark.intel.com/products/52213. Accessed: 3/09/2013.

[71] Borman Jacobson, Braden. TCP Extensions for High Performance. http://tools.ietf.org/html/rfc1323. Accessed: 2/07/2013.

[72] Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. In *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 88–99, New York, NY, USA, 2002. ACM.

[73] Dan Kegel. The C10K Problem. http://www.kegel.com/c10k.html. Accessed: 26/06/2013.

[74] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, Mountain View, CA, USA, 1994.

[75] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Trans. Program. Lang. Syst.*, 12(1):84–101, January 1990.

[76] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[77] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[78] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[79] Doug Lea. malloc - A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html. Accessed: 01/08/2013.

[80] Jonathan Lemon. Kqueue - a generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153, Berkeley, CA, USA, 2001. USENIX Association.

[81] The Linux man-pages project. BSD Socket API - accept(). http://man7.org/linux/man-pages/man2/accept.2.html. Accessed: 26/06/2013.

[82] The Linux man-pages project. BSD Socket API - bind(). http://man7.org/linux/man-pages/man2/bind.2.html. Accessed: 26/06/2013.

[83] The Linux man-pages project. BSD Socket API - connect(). http://man7.org/linux/man-pages/man2/connect.2.html. Accessed: 2/08/2013.

[84] The Linux man-pages project. BSD Socket API - listen(). http://man7.org/linux/man-pages/man2/listen.2.html. Accessed: 26/06/2013.

[85] The Linux man-pages project. BSD Socket API - poll(). http://man7.org/linux/man-pages/man2/poll.2.html. Accessed: 26/06/2013.

[86] The Linux man-pages project. BSD Socket API - recv(). http://man7.org/linux/man-pages/man2/recv.2.html. Accessed: 26/06/2013.

[87] The Linux man-pages project. BSD Socket API - select(). http://man7.org/linux/man-pages/man2/select.2.html. Accessed: 26/06/2013.

[88] The Linux man-pages project. epoll. http://man7.org/linux/man-pages/man7/epoll.7.html. Accessed: 29/07/2013.

[89] The Linux man-pages project. Fast Scatter-Gather I/O - readv(), writev() and iovec. http://www.gnu.org/software/libc/manual/html_node/Scatter_002dGather.html. Accessed: 26/06/2013.

[90] The Linux man-pages project. mmap. http://man7.org/linux/man-pages/man2/mmap.2.html. Accessed: 01/08/2013.

[91] The Linux man-pages project. sched_setaffinity(). http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html. Accessed: 29/07/2013.

[92] Linux man pages. convert(1). http://linux.die.net/man/1/convert. Accessed: 4/09/2013.

[93] Linux man pages. imagemagick(1). http://linux.die.net/man/1/imagemagick. Accessed: 4/09/2013.

[94] B.B. Mandelbrot. *The Fractal Geometry of Nature*. Henry Holt and Company, 1983.

[95] Ashton Mason. The Theron Library - Lightweight Concurrency with Actors in C++. http://www.theron-library.com. Accessed: 3/09/2013.

[96] Friedemann Mattern. Virtual time and global states in distributed systems. In *Proc. Int. Workshop on Parallel and Distributed Algorithms*, pages 215–226, Gers, France, 1988. North-Holland.

[97] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989.

[98] Maged M. Michael. Cas-based lock-free algorithm for shared deques. In Harald Kosch, Lszl Bszrmnyi, and Hermann Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 651–660. Springer, 2003.

[99] Maged M. Michael. ABA Prevention Using Single-Word Instructions. http://www.research.ibm.com/people/m/michael/RC23089.pdf, 2004. Accessed: 04/04/2013.

[100] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[101] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 45–54, New York, NY, USA, 2009. ACM.

[102] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, September 1992.

[103] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, September 1992.

[104] Jens Palsberg. Featherweight x10: a core calculus for async-finish parallelism. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 1–1, New York, NY, USA, 2012. ACM.

[105] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[106] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[107] Sub Ramakrishnan, I.-H. Cho, and L.A. Dunning. A close look at task assignment in distributed systems. In *INFOCOM '91. Proceedings. Tenth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking in the 90s., IEEE*, pages 806–812 vol.2, 1991.

[108] Michel Raynal and Mukesh Singhal. Logical Time: Capturing Causality in Distributed Systems. *Computer*, 29(2):49–56, February 1996.

[109] Jennifer M. Schopf and Francine Berman. Stochastic scheduling. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, New York, NY, USA, 1999. ACM.

[110] Reinhard Schwarz. Causality in Distributed Systems. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, EW 5, pages 1–5, New York, NY, USA, 1992. ACM.

[111] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, 7(3):149–174, March 1994.

[112] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Principles, 7TH Edition*. Wiley student edition. Wiley India Pvt. Limited, 2006.

[113] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[114] D. Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *Software Engineering, IEEE Transactions on*, SE-9(3):219–228, 1983.

[115] Sriram Srinivasan. Kilim. `http://www.malhar.net/sriram/kilim/`. Accessed: 3/09/2013.

[116] The C++ Standard Committee. C++. `http://www.open-std.org/jtc1/sc22/wg21/`. Accessed: 10/08/2013.

[117] Herb Sutter. The free lunch is over. `http://www.gotw.ca/publications/concurrency-ddj.htm`. Accessed: 25/02/2013.

[118] The Transaction Processing Council. Industry Database Benchmarks. `http://www.tpc.org/default.asp`. Accessed: 10/08/2013.

[119] Jan van Leeuwen and Richard B. Tan. Interval routing. *Comput. J.*, 30(4):298–307, 1987.

[120] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. *SIGPLAN Not.*, 36(7):34–43, June 2001.

[121] A. M. Van Tilborg and L. D. Wittie. Wave scheduling decentralized scheduling of task forces in multicomputers. *IEEE Trans. Comput.*, 33(9):835–844, September 1984.

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

[123] James E. White. Remote Procedure Calls - Initial Proposition 1976. http://tools.ietf.org/html/rfc707. Accessed: 10/07/2013.

[124] yWorks. The yEd Graph Editor. http://www.yworks.com/en/products_yed_about.html. Accessed: 5/09/2013.