San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Spring 2012

Actor-based Concurrency in Newspeak 4

Nikolay Botev
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

Recommended Citation

Botev, Nikolay, "Actor-based Concurrency in Newspeak 4" (2012). *Master's Projects*. 231.
https://scholarworks.sjsu.edu/etd_projects/231

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Actor-based Concurrency in Newspeak 4

A Project Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Computer Science

By

Nikolay Botev

May 2012

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Jon Pearce

Dr. Gilad Bracha

Dr. Cay Horstmann

APPROVED FOR THE UNIVERSITY

ABSTRACT

ACTOR-BASED CONCURRENCY IN NEWSPEAK 4

by Nikolay Botev

Actors are a model of computation invented by Carl Hewitt in the 1970s. It has seen a resurrection of mainstream use recently as a potential solution to the latency and concurrency that are quickly rising as the dominant challenges facing the software industry. In this project I explored the history of the actor model and a practical implementation of actor-based concurrency tightly integrated with non-blocking futures in the E programming language developed by Mark Miller. I implemented an actor-based concurrency framework for Newspeak that closely follows the E implementation and includes E-style futures and deep integration into the programming language via new syntax for asynchronous message passing.

TABLE OF CONTENTS

1 Introduction
2 Motivation
2.1 The Actor Model
2.2 Pure Actor-based Languages
2.3 Critiques of Actors
2.4 Advantages of Actors
2.5 Criteria for a Good Actor-based Concurrency Implementation
3 Related Work
3.1 Asynchrony in Mainstream Java Frameworks
3.2 Mainstream Actor Frameworks
3.3 Fringe Alternatives
3.4 Conclusion
4 Overview of Newspeak 4 Actors
4.1 Carl Hewitt's Actor Model
4.2 E-style Actors in Newspeak 4
4.3 Comparison to Java Threads
4.4 Actors as a Method of Execution Resource Management
4.4.1 Memory Management
4.4.2 Execution Resource Management
4.4.3 Conclusion
5 Newspeak 4 Actors by Example
5.1 A Gentle Introduction to Newspeak
5.2 Asynchronous Factorial
5.3 Subtleties of Asynchrony
5.4 Transparent Potential for Parallelism
5.5 Patterns for Asynchronous Programming
5.5.1 Using Promise Pipelining
5.5.2 Using whenResolved:
5.5.3 Dealing with Loops
5.5.4 Waiting for Several Results
5.5.5 MapReduce
5.6 Publisher/Subscriber
5.7 Breadth-first Tree Walking Using Recursion
5.8 Modules, Mutual Recursion and Actors
6 Functional Specifications
6.1 Introduction
6.2 Messages
6.2.1 Immediate-send
6.2.2 Eventual-send
6.3 Activation Stack
6.4 Message Queue
6.5 Heap
6.6 Actor Communication
6.7 Reflection and Actors
6.6.1 Introduction to Mirrors
6.6.3 The Reference Mirror
6.6.2 The Actor Mirror
6.6.3 Eventual-send Proxies
6.8 Argument-passing Semantics
6.8.1 Pass-by-Far-Reference
6.8.2 Pass-by-Value
6.8.3 Far Reference Passing Semantics

6.9 The Glue Holding Actors Together
6.10 Eventual-sends
6.11 Actor Lifecycle
6.12 Actor Creation
6.13 Bootstrapping Newspeak
6.14 Actor Termination
6.15 Futures in Newspeak
6.15.1 Promise Listeners
6.15.2 Exceptions
6.15.3 Promise Sequencing
6.15.4 Promise Pipelining
6.15.5 Promises and Double Dispatch
6.15.6 The Promise Protocol
6.16 Order of Message Delivery
6.16.1 Non-determinism in E-ORDER
6.16.1.1 Scenario 1
6.16.1.2 Scenario 2
7 Design and Implementation
7.1 Design
7.1.1 Overview
7.1.2 Actor System Portability
7.1.3 Actor System Module Interactions
7.1.4 GUI as an Actor
7.1.5 Pastime and Past
7.2 Implementation
7.2.1 The Dispatcher
7.2.2 The Actor Class
7.2.3 Argument Passing
7.2.4 Promise resolution
7.2.5 Value objects
7.2.6 Asynchronous Control Structures
8 Conclusion
References
Appendix I - Scope and Limitations of the Current Implementation
Appending II - Bootstrapping the Actor Framework

1 Introduction

Newspeak is a modular, object-oriented programming language. From the Newspeak home page [1]:

> Like Self, Newspeak is message-based; all names are dynamically bound. However, like Smalltalk, Newspeak uses classes rather than prototypes. As in Beta, classes may nest. Because class names are late bound, all classes are virtual, every class can act as a mixin, and class hierarchy inheritance falls out automatically. Top level classes are essentially self contained parametric namespaces, and serve to define component style modules, which naturally define sandboxes in an object-capability style.

In its current version -- Newspeak 3, the language has no support for concurrent programming. The goal for Newspeak from its inception is to eventually provide first-class support for Actor-based concurrency with E-style non-blocking futures [2]. This thesis describes the motivation, specification, design and implementation of an Actor-based concurrency framework built for the next version of the Newspeak language -- Newspeak 4.

We begin Chapter 2 with an overview of the actor model as a theory of computation. Chapter 3 details the motivation for the choice of actors as opposed to more traditional approaches to concurrency, including the definition of a set of criteria for a good actor framework and a brief evaluation of existing concurrency frameworks based on these criteria. Chapter 4 provides a high-level overview of actors in Newspeak 4. Then in Chapter 5 we introduce Newspeak 4 actors by example, highlighting a number of patterns of actor usage. Chapter 6 discusses the core architecture of the actor framework in Newspeak 4, followed by a brief discussion of the integration of actors with the Newspeak reflection facilities, the semantics of actor communication through far references, actor lifecycle, futures, and an examination of the order of actor message delivery. Chapter 7 discusses the design and implementation of the Newspeak actor framework and finally, Chapter 8 concludes with an optimistic outlook for the future of actors.

2 Motivation

The many-core problem [3] is well understood and has been widely discussed over the past several years [4, 5, 6, 29]. The trend of flattening out single-thread performance and increasing number of cores per CPU is pushing the limits of the traditional approach to concurrency, namely that of using shared state (memory) for communication among multiple independent threads of execution. Shared-state concurrency suffers from a number of problems, such as deadlock, livelock, race conditions etc., in addition to a difficulty in scaling out to large datasets. All of these problems are well understood[1] and a detailed discussion is beyond the scope of this report.

An alternative approach to shared-state concurrency is message passing without shared state. An example of an important modern application based on message passing is Google MapReduce -- a model designed to simplify computation on large clusters [7]. Open-source implementations of Google MapReduce, such as Hadoop have seen large adoption over the past several years [8] and play an important role in dealing with Big Data that is spread across many nodes on a cluster. A key aspect of MapReduce is that code runs on the nodes where the

[1] Chapter 13 of [13] contains a thorough examination of the problems of shared-state concurrency.

data resides and that data is sent between nodes for further processing, i.e. the distribution of the data among nodes in the cluster is explicit and MapReduce is inherently a greedy approach to computation, based on the classic divide-and-conquer technique [7].

2.1 The Actor Model

The Actor model is a theoretical model of computation developed by Carl Hewitt that is exclusively based on asynchronous message passing between primitives called “Actors” [9]:

> An Actor is a computational entity that, in response to a message it receives, can concurrently:
> - send messages to other Actors;
> - create new Actors;
> - designate how to handle the next message it receives.

The actor model stands at the same level of abstraction and can be contrasted to other models of computation such as the Turing machine and the von Neumann machine.

In the author’s opinion, it is crucially important to understand that the Actor model as a theoretical model captures the essence of message passing as an abstraction and this abstraction must be kept separate from the specific implementation details of any actor-inspired concurrency framework. In Hewitt’s own words, “[t]he following are not required by an Actor: a thread, a mailbox, a message queue, its own operating system process, etc” [9]. For example, while many actor-based concurrency frameworks (for example Akka [10]) employ one message queue per actor, this is not strictly necessary -- an actor framework could conceivably share the same message queue among many actors.

2.2 Pure Actor-based Languages

A pure actor-based language is one, which expresses computation exclusively via the use of actors. A small number of such languages exist, such as SAL [11] and Humus [12]. The fundamental primitives in a pure actor-based language closely mirror the definition of an Actor quoted in the previous section:

- SEND - send a message to an actor
- NEW - create a new actor
- BECOME - designate how the current actor should handle the next message

In practice, however, pure actor-based languages also include a primitive for pattern-matching on the value of an incoming message. For example, a part of the implementation of a future actor in Humus looks like this [14]:

```
LET future_beh = \msg.[
      CASE msg OF
      (cust, #write, value) : [
            BECOME value_beh(value)
            SEND SELF TO cust
      ]
      (cust, #read) : [
            BECOME wait_beh(cust)
      ]
      END
```

```
]
```

In the code above, LET binds the identifier future_beh to the behavior definition that follows the equal sign, and “CASE msg OF …” is the pattern-matching primitive, which is used to select the code to execute based on the contents of the incoming message msg. A plausible alternative approach to the design of a pure actor language is to model actors as objects, and rely on object-oriented virtual message dispatch (polymorphism), eliminating the need for a pattern-matching primitive. In the example above, the behavior of future_beh would then be modeled as an instance of a class with two methods -- write and read, which accept arguments (cust, value) and (cust) respectively. In this sense, a pure actor system can be viewed as a collection of objects (actors), which communicate exclusively via asynchronous one-way message passing.

2.3 Critiques of Actors

There are two main critiques of actors and other message passing systems -- inversion of control, and the performance overhead of message passing. Inversion of control refers to the break of control flow that occurs in pure one-way message passing systems. If function A calls, or waits for B and then does something with the result in a traditional shared-memory system, then to implement the same behavior in a pure one-way message passing system, A and B must be rewritten such that A is split into two parts -- A’, which contains the first part of A up to the point of calling B, and A’’ -- the second half of function A, which processes the result of B. Then B must be rewritten to explicitly send a message back to A’’, instead of directly returning its result to A as in the shared-memory system case. This transformation is practically equivalent to continuation passing style, and indeed this is how pure actor-based languages such as SAL and Humus deal with control flow. The common pattern in such languages is for A’ to create a customer actor representing A’’ (the remainder of the computation) and pass that actor as a parameter to B, which then sends a message back to the A’’ actor with the result of its computation.This style of programming is indeed cumbersome -- it adds more burden to the programmer and makes code harder to read, understand and maintain. The inversion of control problem can be addressed via the use of E-style futures, which are described in detail later in this report.

The performance overhead of message passing refers to the need to copy data that is passed in messages between actors. In a hypothetical example, if we have a large data structure representing a jumbo jet airplane, which needs to be accessed and manipulated by several actors, then the performance overhead of passing the jumbo jet data structure between actors can become prohibitively expensive. This problem is typically addressed in two ways in actor systems. The first is to create a single actor responsible for manipulating the large data structure and encapsulate all code, which accesses and manipulates it in this actor. All other actors, which need to initiate some manipulation on the jumbo jet would then delegate their work by sending a message to the jumbo jet actor, who then safely performs the manipulation. A practical example of this approach is a Graphical User Interface (GUI) system - an application’s GUI can consist of multiple top-level windows, each consisting of hundreds or even thousands of GUI widgets. The GUI is best represented as a single actor, which takes full responsibility of maintaining and manipulating all of the windows, widgets and other graphical objects that comprise the GUI system. The single actor pattern in effect enforces a serialization of all operations on the large data structure it maintains, which is the logical equivalent of a lock in a shared memory system.

The second way to address the overhead of message passing when manipulating

large volumes of data is via the use of immutable data structures. A good actor system implementation, which runs on hardware optimized for shared-memory concurrency can avoid the expensive copy operation when passing immutable data between actors that are running within the same address space. Since the data structure is immutable, such sharing is transparent to the running application and does not affect the runtime semantics of actors. Efficient algorithms for immutable data structures, which can avoid making a full copy of the data exist, and can be used to further alleviate the cost of manipulating large data sets [15]. A discussion of such algorithms is beyond the scope of this report.

2.4 Advantages of Actors

The first and primary advantage of actors is that since Actors execute concurrently and do not share state, but instead communicate exclusively via message passing, an actor-based concurrency framework does not suffer from any of the problems of shared-state concurrency, such as deadlocks.

Actors are a natural fit for implementing frameworks such as Google MapReduce. Each pieces of a MapReduce job [7] can be modeled as an actor performing the map or reduce phase of the computation on its local subset of the data, and communicating the results in a message to another actor responsible for the next computation phase. In this sense, actors are greedy by design and can naturally lead towards a divide-and-conquer solution to the problem at hand. This is the second main advantage of actors over shared-state concurrency -- adaptive concurrency. A good actor implementation allows a computation problem to be expressed in many fine-grained actors, and the actors can then be scheduled efficiently and transparently on the number of execution resources available in the system, whether it is a small 2-core laptop, a large SMP box with multiple CPU sockets and cores, or a large-scale cluster with hundreds of interconnected computer nodes. Carl Hewitt calls this iAdaptive concurrency and defines it as “the ability to express computations that can be adapted s to fit available resources in terms of demand and available capacity” [9].

2.5 Criteria for a Good Actor-based Concurrency Implementation

Implementing a good actor-based concurrency framework is hard. As we saw in the previous sections, solutions to the commonly cited drawbacks of actor systems and the ability to take full advantage of the benefits of actors depend on the availability of certain infrastructure and characteristics of the actor system implementation. This section aims to lay out a concrete set of criteria for the design and implementation of a high-quality actor system. We identify the following set of high level criteria:

1. No blocking operations
2. E-style (integrated and non-blocking) Futures
3. Automatic actor lifecycle
4. Low per-actor overhead

Criterion 1 refers to the full absence of blocking operations on actors, i.e. operations, which block an actor from receiving and processing further messages. The two most common examples of such an operation, are blocking call and blocking receive. A blocking call is an operation, which sends a message to an actor and blocks the current actor until a response is received. A blocking receive is an operation, which blocks the current actor, waiting for a message of a certain type to arrive, suspending the processing of messages, which do not match the type(s) of messages being awaited. Blocking operations introduce the possibility of deadlocks and therefore their presence eliminates one of the main advantages of actor-based

concurrency over shared-state concurrency. In an actor framework without blocking operations, problems that require “blocking” are solved via the explicit use of insensitive actors, a pattern described in [11].

Integrated futures (criterion 2) provide for fully-functional asynchronous two-way communication between actors, i.e. a non-blocking way to receive the result or exception of an asynchronous message. Without integrated futures, actors suffer from inversion of control and control flow must be expressed via continuation-passing style, which requires extra development effort and results in code that is more difficult to read and maintain. Error handling also becomes more difficult. Erlang-style actors [27], for example, resort to supervisor hierarchies -- a pattern, which provides orthogonal error handling via linked actors called supervisors. Supervisors trap all errors that occurs in other actors. This a good pattern, which alleviates the need for custom error-handling boilerplate code, but error handling via supervisors is orthogonal to the control flow of the program and still suffer from inversion of control. E-style futures propagate exceptions in a manner that is equivalent to the way exceptions are propagated along the call stack in traditional sequential computation. In order to fulfil criterion 1, futures must also be non-blocking and therefore code that receives a message response can only be scheduled asynchronously. In order to properly integrate into an actor-based concurrency implementation, futures must use actors as the means of scheduling code for asynchronous execution. This implies that Futures must be tightly coupled to the internal actor message dispatch mechanism of the implementation.

The last two criteria -- automatic actor lifecycle and low per-actor overhead are required in order to alleviate the developer from the burden of worrying about the number of actors created and the need manual performance tuning configuration depending on the number of execution resources available in the hardware (CPUs, cores, hardware threads etc). Automatic actor lifecycle means that an actor is not tied to any operating system resource such as a process, thread, fiber etc, and all of the actor’s resources (heap, mailbox, stack etc) are automatically reclaimed as soon as all references to that actor go away, and the actor’s mailbox becomes empty. The per-actor overhead must be low enough such that there is no need for creating actor pools. Resource pools (thread pools, memory pools, connection pools etc) are typically created to manage and reuse resources that are expensive to create and destroy. An actor-based concurrency implementation is very likely to internally make use of such resource pools, but must hide their presence by decoupling individual actors from the underlying expensive resources, and keep the costs of creating and destroying actors to a minimum. This ensures that the actor implementation can provide iAdaptive concurrency, which is one of the key advantages of actors.

3 Related Work

This chapter presents a brief survey and evaluation of related work on asynchronous message-passing-based concurrency in terms of the criteria outlined in the previous section.

3.1 Asynchrony in Mainstream Java Frameworks

In early versions of Java application servers -- EJB up to version 3, the only means for asynchronous communication is via JMS beans. JMS beans are heavyweight -- each bean is tied to a Java thread, and there is a blocking send operation. Beans are created in pools via XML or annotation-based configuration, and there is no integration of non-blocking futures. JMS does not meet any of the criteria for a good message-passing framework. Starting with

EJB 3.1, session EJBs can contain methods that are invoked asynchronously, via the use of the @Asynchronous annotation. Asynchronous EJB methods that return a value must use java.util.concurrent.Future, which only provides a blocking get API for retrieving the future's result. Session EJBs also suffer from the same lifecycle deficiencies as JMS beans - configuration-based creation and pooling. Prior to EJB 3.1, application servers such as JBoss provide asynchronous extensions to the EJB specification. The JBoss extensions also rely exclusively on java.util.concurrent.Future and therefore suffer from the same limitations as the EJB 3.1 standard.

The Spring Framework, which is a popular alternative to EJB standard-based application servers, also provides for asynchronous method invocation on Spring beans via an @Async annotation. Spring @Async also relies exclusively on the blocking java.util.concurrent.Future for passing return values. Spring beans are relatively lightweight objects, but are typically singletons that are always assigned a unique name and registered in a global bean factory and live for the entire duration of the application. Therefore Spring beans do not meet the lifecycle criteria either.

3.2 Mainstream Actor Frameworks

Scala actors [28] are inspired by erlang actors and are very lightweight and have a semi-automatic lifecycle. However, as in erlang, Scala actors support a blocking receive API. Scala actors include a fully integrated Future API, however scala actor futures include a blocking retrieve operation. While it is possible to program scala futures in a purely non-blocking fashion [22], it can be difficult to do so correctly and especially so when programming asynchronous control flow.

Scalaz actors [30] are lightweight but provide for one-way message-passing only. The included scalaz Future API is not integrated with scalaz actors, as scalaz futures schedule their non-blocking operations outside of the context of actors. This is a surprisingly recurring pattern across actor frameworks for programming languages that execute on the Java Virtual Machine. Lift Actors, the GPars groovy parallels library, Akka, and Twitter's Finagle framework all include an actor framework along with a futures API, which is supposed to be used with actors, but schedules code for execution outside of the context of an actor. The problem with this approach is that code scheduled by a Future cannot safely access any actor's state. All of the aforementioned frameworks also include a blocking API for retrieving a future's result.

3.3 Fringe Alternatives

This section evaluates a collection of mostly academic efforts in the area of actors, which were surveyed in [16].

Kilim is not strictly an actor framework since it employs the use of explicit channels, which can be of a bounded size and therefore can block. Kilim does not include a Future API for asynchronous control flow. ActorFoundry, Actor Architecture and AJ all include a blocking call API as the sole means for asynchronous control flow. JavAct includes a blocking Futures API via its Future.getReply() method. Actors Guild includes a Future API in AsyncResult, which supports non-blocking result handling but is not integrated and schedules non-blocking code outside of the context of an actor. Actor Guild's AsyncResult also includes a blocking get() method.

Jsaab, which is not strictly an actor framework, provides higher-level asynchronous primitives such as publisher-subscriber channels with multiple producers and consumers very similar to

JMS. Two-way communication uses a blocking request/reply API. Jetlang is also not an actor framework, since it employs separate notions of channels for communication, and fibers for execution. Fibers are lightweight and channel communication is fully asynchronous, however without the presence of non-blocking futures for asynchronous control flow, even simple examples in Jetlang are counter-intuitive and hard-to-understand.

SALSA -- a pure actor-based language, supports the use of tokens for asynchronous control flow. Tokens are similar to but more restrictive than futures -- tokens cannot be used as return values themselves, and cannot accept messages. SALSA actors are heavyweight because each actor executes in a separate a Java thread [16].

3.4 Conclusion

Judging based on the results of the above survey alone one might jump to the conclusion that it must be impossible to implement an actor framework that meets all of the criteria outlined earlier in this report. Yet such an implementation does exist in the E language and its E-on-Java implementation, and BootComm system. E futures provide an exclusively non-blocking API and are properly integrated with E vats (actors), and E vats running within the same JVM are lightweight -- multiple vats can share a single Java thread and associated message queue. E vats have an automatic lifecycle as well. AmbientTalk is another language with an actor-based concurrency framework that is directly inspired by E and meets all of the criteria outlined here. Actors in Newspeak 4 also closely follow the design of E, and attempt to take its ideas further by reducing the per-actor overhead to the greatest extent possible and providing a very simple actor creation API.

4 Overview of Newspeak 4 Actors

Newspeak Actors are similar to OS threads in the sense that each actor executes in its own logical thread of control. A Newspeak 4 program consists of one or more actors at runtime, just like a Java program consists of one or more threads. Just as a Java program begins its execution with one main thread, a Newspeak 4 program begins its execution with one main actor.

There are a few differences between the original Actor model and Actors in Newspeak that we outline in the following sections.

4.1 Carl Hewitt's Actor Model

The original Actor model is purely functional – every actor consists of two parts -- a function, and a mailbox. The function defines the actor's behavior and the mailbox receives pending asynchronous invocation requests of the function. The behavior of an actor can be mutated, by assigning a new function to take over subsequent message processing (invocations), via a built-in become statement. All functions in that model are only used as part of actors, including primitive functions such as number addition, multiplication etc. Computation bottoms out at these primitive functions, and control flow can be expressed via futures and/or continuation passing style. The order of message delivery is undefined. The only guarantee is that every message sent will eventually be delivered at some indeterminate point in the future.

4.2 E-style Actors in Newspeak 4

Actor-based concurrency in Newspeak 4 differs from the original Actor model and pure actor-based languages such as SAL and Humus in the following ways:

1. In Newspeak, an actor's mailbox is explicitly defined as a (virtual) queue, and there is a well-defined order of message delivery (E-ORDER). This is in contrast to Hewitt's Actor model, where message delivery order is undefined.
2. Actors in Newspeak are object-oriented. That is, each actor message, instead of representing a function call, represents a message send to an object. In other words, each message contains an implicit argument representing the receiver object and message dispatch in an actor relies on OO method dispatch versus pattern matching in SAL/Humus.
3. Actors in Newspeak are not "pure." In a "pure" actor-based language every object is an actor, and therefore objects communicate exclusively via asynchronous message passing. Newspeak 4 is a hybrid system, in the same way as the E language. An actor encapsulates a collection of objects with an associated mailbox and activation stack. All objects of an actor then share the same mailbox and activation stack, and can communicate with each other synchronously, just as in traditional object-oriented programming languages. Each object belongs to one and only one actor for the object's entire lifetime.

4.3 Comparison to Java Threads

The simplest way to model a Newspeak Actor in concrete Java terms is as a Java thread with an associated message queue, which runs in a loop taking messages from the queue and processing them one at a time:

```
class NaiveActor extends Thread {
  private final Queue queue;
  public void run() {
    while (true) {
      Message msg = queue.dequeue();
      msg.method.invoke(msg.receiver, msg.arguments);
    }
  }
}
```

The above class is an oversimplification and misrepresentation in a number of ways, however it does highlight the following important features of actors:

1. All objects belonging to an actor share the same queue.
2. All objects belonging to an actor share the same activation stack, just like all method calls within a single thread in Java share the same call stack.

Newspeak 4 Actors differ from this example in the following important ways:

1. Every object belongs to one and only one actor for the entire duration of its life span. This is unlike in Java, where threads can share direct access to the same object.
2. Synchronous communication is only possible between objects belonging to the same

actor.

3. The above `NaiveActor` class omits the logic for returning a result from an asynchronous message, which is supported via the integration of E-style futures in Newspeak.
4. Actors are much more lightweight than Java threads. Java threads map directly to OS threads, and carry the associated scheduling and OS resource overhead. Actors are lightweight – the overhead of a Newspeak 4 actor is the same as that of a small object.
5. Actor lifecycle is managed automatically, whereas threads have to be explicitly started and stopped (when using thread pools).
6. In normal operation (the reflection use case aside), there is no object representing an actor as a whole akin to a Java Thread object (in the reflection use case an actor mirror object can be created). There is no direct access to the actor's queue, and only cross-actor object references contain the ability to mutate the queue of their target object's actor. References to objects within the same actor, can also be used to communicate asynchronously with their target object, and therefore can mutate the currently executing actor's queue.

4.4 Actors as a Method of Execution Resource Management

Actors are fundamentally an abstraction of the execution resources (CPUs) of the system. An actor in Newspeak 4 is the unit of allocation of the execution resources on the system. At its core, a language runtime environment (such as a Java/.Net/Javascript/Squeak/Dalvik/Dart VM) provides two types of resources – **execution resources** (CPU cores/hardware threads) and **memory** (RAM) and provides facilities for the management of these resources. In this section, we will compare and contrast the Newspeak 4, Java and C programming languages in terms of their facilities for execution resource and memory management.

4.4.1 Memory Management

In Java memory is managed via object construction (new). Compared to C, memory management in Newspeak and Java is **simple**, **lightweight** and **automatic**[2]. It is simple, because there is only one way to allocate memory – object construction, compared to the variety of APIs available in C, such as malloc, alloca etc. It is lightweight (at least in the common case for small short-lived objects), because the cost of an object allocation ("new ...") in Java is lower than the cost of a malloc in C [17]. It is automatic, because object lifecycle is managed by the runtime and objects are automatically freed when no longer used via garbage collection – there is no need for memory bookkeeping code in the application, having to call free() in order to release memory at the right time in the right place.

Garbage collection in Newspeak or Java, and explicit memory management in C are two alternative paradigms for memory resource management with fundamentally different characteristics.

4.4.2 Execution Resource Management

The story of execution resource management in Java is different. Java execution resource management is practically the same as that in C. At a high level Java threads are not much different from POSIX threads available in C and C++. Both Java threads and posix threads map directly to Operating System (OS) threads on all modern operating systems, including

---

[2] Both Newspeak and Java derive their memory management model from Smalltalk, where it is referred to as "automatic storage management." Today this type of memory management is often referred to as garbage-collected memory.

Windows, OS X, Linux and Solaris. OS threads are neither **simple**, nor **lightweight**, nor **automatic**. OS Threads are not simple, because efficient thread management requires the careful allocation of threads in thread pools, explicit coordination of shared data access via locks, and communication among threads is managed via a multitude of non-trivial mechanisms such as conditions, barriers, phasers (see [18] for a case in point), blocking queues etc. OS Threads are not lightweight, because thread creation is a relatively expensive operation as it requires the allocation of a limited OS resource -- a thread handle, and the allocation of a fixed-size thread stack, and other thread-local structures in the operating system kernel. OS Threads are not automatic, because their lifecycle is manually managed by the programmer – an OS thread must be explicitly started via its start method, and potentially explicitly stopped if part of a thread pool – additional complications arise from the different lifecycle semantics of daemon threads versus regular threads.

Newspeak 4 actors stand in stark contrast to Java and C threads in that actors share all of the characteristics of Newspeak and Java memory management described earlier -- actors are simple, lightweight and automatic. Actors are simple, because there is only one way to schedule an execution – via an actor, and since actors are lightweight (as explained later) there is no need for explicit pooling of actors. Actors are simple, also because there is no need for explicit coordination of shared state access and the associated complexities that come with that. Finally, actors are simple because there is only one way of communication between actors, and that is via asynchronous message passing between actors' objects. Actors are lightweight, because creation of an actor does not require the explicit allocation of any OS resources, such as a process, thread or stack. Both the actor's heap and queue can be virtual, in the sense that multiple actors can internally share the same queue. Actors running within the same address space can share the same heap, with actor heap state isolation enforced by the message passing infrastructure (which can rely on thread-safe pointer manipulation [13]).

Actors in Newspeak 4 and Java or C threads are two alternative paradigms for execution resource management with fundamentally different characteristics.

4.4.3 Conclusion

The analysis in this section demonstrated how Newspeak 4 actors compare to Java and C threads , just as garbage collection in Newspeak and Java compares to explicit memory management in C. Actors provide the same advantages for execution resource management that garbage collection provides for memory resource management. Just like garbage collection, actors are simple, lightweight and automatic. This is in contrast to OS threads and explicit memory management, which provide none of those three qualities. Just like garbage collection eliminates the need for most memory resource (object) pooling (assuming a high-quality garbage collector implementation), actors eliminate the need for most execution resource (OS thread) pooling (assuming a high-quality actor implementation).

5 Newspeak 4 Actors by Example

This chapter introduces Newspeak 4 actors by example. We begin with a brief introduction to Newspeak syntax, then introduces the various actor features with simple examples, which are followed by recipes for transforming sequential code into asynchronous. The last few examples then demonstrate basic patterns for actor programming.

5.1 A Gentle Introduction to Newspeak

Let's begin with a simple example -- the factorial function implemented in Java:

```
public int factorial(int n) {
  if (n > 1) return n * factorial(n - 1);
  else return 1;
}
```

The equivalent function in Newspeak looks like this:

```
public factorial: n <Integer> ^ <Integer> = (
  (n > 1) ifTrue: [ ^n * factorial: (n - 1) ]
  ifFalse: [ ^1 ].
)
```

Note a few differences in Newspeak syntax. The type annotations (`<Integer>`) for the argument `n` and the return value are enclosed in angle braces and follow, instead of preceding, their target. This is similar to languages like Pascal as opposed to the Java syntax for type annotations, which is in the C tradition. The method body is enclosed in regular parentheses instead of curly braces. The hat (`^`) character is the equivalent to the return keyword in Java. A more significant difference is in the if statement, which in Newspeak is expressed as a message send[3] (method call) to the boolean expression `(n > 1)`. In this case the message selector (method name) is `ifTrue:ifFalse:` and takes two code blocks (closures) as arguments. Code blocks are enclosed in square brackets. Only one of the code blocks will be evaluated depending on the value of the boolean expression, resulting in the same semantics as those of an if expression in Java.

5.2 Asynchronous Factorial

The factorial implementation above can be rewritten in an asynchronous fashion with only one very small modification:

```
public factorial: n <Integer> ^ <Promise[Integer]> = (
  (n > 1) ifTrue: [ ^n <-: * factorial: (n - 1) ]
  ifFalse: [ ^1 ].
)
```

The only change in the code above is the addition of the eventual-send operator `<-:` to the recursive case (`ifTrue:`) of the branch. With this change the factorial function (except for the bottom case where `n <= 1`) will return a promise for the result of its computation, instead of blocking the caller until the function computes the result. Note that in this example there is no explicitly added concurrency or parallelism to the internal operations involved in the factorial computation itself. There is external concurrency, however, in that the computation of the factorial will proceed concurrently (although not in parallel) with other processing done by the same actor.

The primary purpose of this example is to illustrate the common pattern of transforming sequential code to asynchronous code in Newspeak. A primary goal of Newspeak 4 actors is to make the task of transforming sequential code into asynchronous and/or parallel code as simple and straightforward as possible. The example above demonstrates the simplest possible

---

[3] Newspeak, like Smalltalk, has no built-in control structures. All control structures, including loops and branches in Newspeak are expressed as message sends.

scenario and how easy it is to convert code to asynchronous execution. The code still looks and feels natural as the intent of the program is expressed in the same was as in the sequential case. There are no special methods, or explicit creation of heavy-weight custom objects such as tasks, queues, channels, futures, pipes, processes, threads, thread pools, executors etc, typical of other asynchronous libraries. There are no new APIs to learn. The only new concept is an eventual-send operator `<-:`, which expresses the explicit intent to schedule an individual computation for eventual (asynchronous) instead of immediate execution. This operator is built into the language and can be used universally when sending any message to any object. The eventual-send operator is a first-class citizen of the language and therefore so is asynchronous computation. There is no need for a special distinction between Actor objects, Enterprise Objects, Enterprise Beans, etc versus regular objects. Every value in Newspeak is a first class object, including primitives such as numbers.

5.3 Subtleties of Asynchrony

The above example is interesting in that the multiplication operator is applied asynchronously, but the recursive call to the factorial function is still synchronous. Even in a simple example as this, we had several choices of where to insert asynchrony. In addition to asynchronously scheduling the multiplication operation, we could have also asynchronously scheduled the recursive call to the factorial function:

```
public factorial: n <Integer> ^ <Promise[Integer]> = (
  (n > 1) ifTrue: [ ^(self <-: factorial: (n - 1)) <-: * n ]
  ifFalse: [ ^1 ].
)
```

Note that we had to reverse the order or the operands to the multiplication function, so that n is the argument of the multiplication message, instead of the receiver. This is because in Newspeak (as in E), primitive operations such as number multiplication cannot accept unresolved promises as arguments[4]. In this example n is a number that is expected to be an immediately available value and can therefore be safely passed as an argument to the multiplication message.

Another important point is that once we transform the recursive call to an asynchronous eventual-send, we end up with a promise for the result of the recursion, and we have no choice, but to eventual-send the multiplication message as well. This is because promises can only accept eventual-sends and therefore eventual-sending the factorial message while immediate-sending the multiplication message is not an option. The result of an eventual-send expression can only be the receiver of other eventual-sends[5].

5.4 Transparent Potential for Parallelism

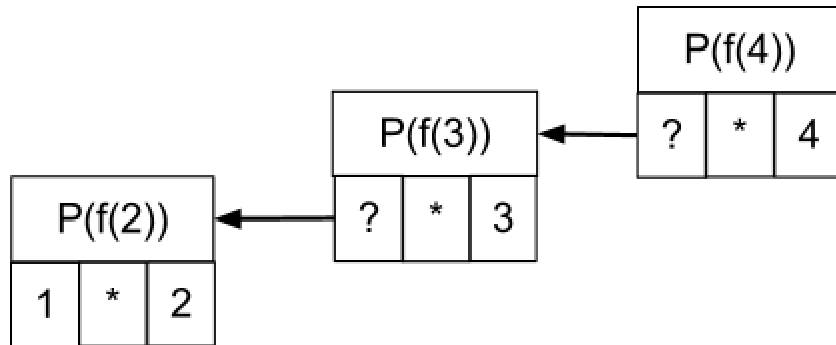
The last, fully asynchronous example of our factorial function is arguably too complex, without adding enough benefit. Multiplication is the only fundamental operation performed by the factorial function and thus scheduling the multiplication asynchronously should be enough. Since multiplication is commutative, computation of the factorial function is inherently

---

[4] Multiplication and other arithmetic operations on primitive types in Newspeak are double-dispatched. Double dispatch, however, always uses immediate-sends, which precludes the use of unresolved promises as arguments in this case. A more detailed discussion of this topic is provided in Section 6.15.5.

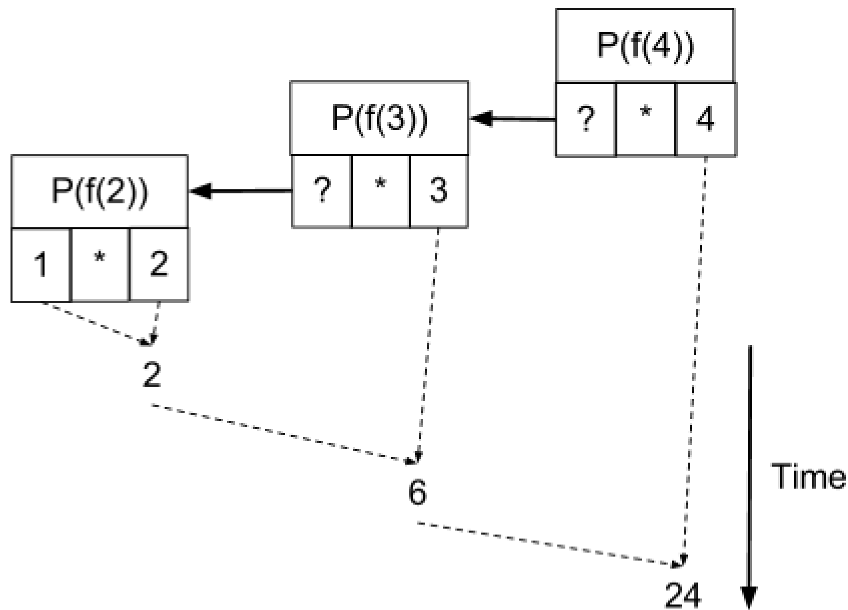
[5] A means for extracting the value of a promise is given later in this chapter.

parallelizable. The question arises then, how can we exploit this inherent parallelism in the context of actor-based concurrency. In the first asynchronous factorial example, an execution of `factorial: n` will in fact synchronously recurse all the way down to `n = 1`, but instead of performing all multiplication operations along the way, the function will simply schedule those operations for later execution and return a promise for the final combined result. Graphically, the result of the execution of `factorial: 4` can be represented as follows:



Each box represents a promise object created as a result of an eventual-send of the multiplication message. The label in the top half of each box represents one specific invocation of factorial. For example P(f(4)) represents the promise for the result of computing the factorial of the number 4 and is the result of invoking factorial: 4. Each of the promise objects additionally holds a buffered message (represented by the 3 boxes underneath the main box) enqueued on the actor's message queue to be processed once the promise value (denoted by ?) gets becomes available.

The simplest way to process these messages is the default mechanism of sequential execution already described: each message is dequeued and processed one by one, as depicted below:

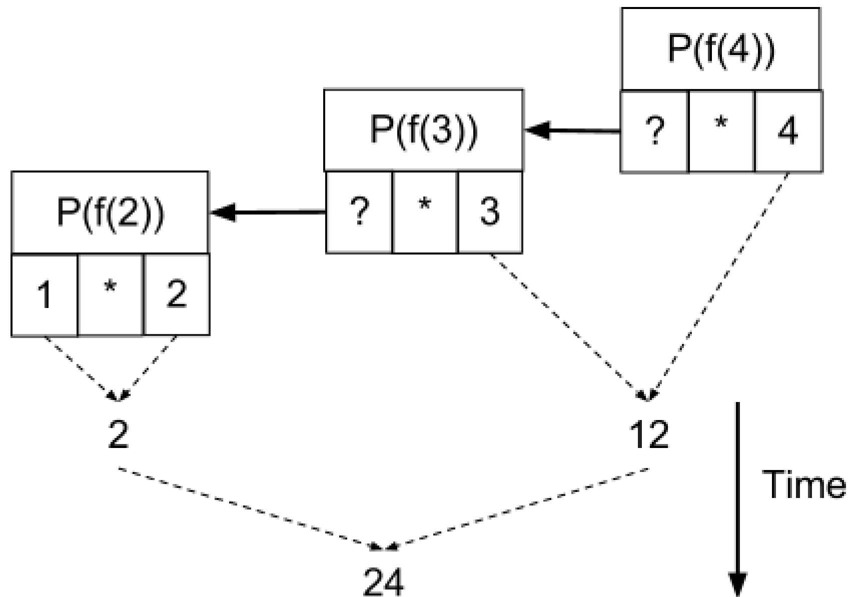


While the above method works, it is sub-optimal. We can do better by exploiting the fact that the multiplication function is commutative and each of the operands is a known immutable value[6].

---

[6] Numbers in Newspeak are value objects. The notion of a value object is described in Section 6.8.

By exploiting this inherent parallelism we can compute the factorial of 4 in two steps instead of three:



The above parallelism can be detected and exploited by a smart actor implementation[7] in a manner fully transparent to the programmer. Note that the factorial function was implemented in the most intuitive and simple way, by encoding the function's mathematical recursive definition in the syntax of the Newspeak language. No optimizing transformations were required to explicitly enable parallel execution. Such transformations obscure the original intent and make code more difficult to maintain.

To achieve the same in Scala, for example, one can use the parallel collections library. The factorial function can then be computed by explicitly applying the reduce operator on the sequence of numbers from 1 to n:

```
def factorial(n: Int): Int = (1 to n).par.reduce(_ * _)
```

The above code, however, encodes the understanding that the factorial function can be expressed as a reduce operation on a sequence of numbers. The original recursive (and presumably intuitive) definition of the factorial function is completely lost. Scala is able in certain cases to transform tail-recursive functions into operations on parallel collections. However, this is still not fully transparent to the user, the function definition must be transformed into its tail-recursive form by carrying over the result in an explicit additional parameters.

One could speculate that potential for parallelism in a chain of promises (such as in the example described in this section) can be detected and exploited efficiently and transparently at runtime and thus provide benefits comparable to those of the scala parallel collections library but at a lower cost to the programmer. This is not implemented in the Newspeak 4 actor framework

[7] Actors in Newspeak 4 provide concurrency but not parallel execution. The parallelization scheme described in this section is not implemented in Newspeak 4.

and the author is not aware of other work in this area. Thus, this is potentially an open area for research.

5.5 Patterns for Asynchronous Programming

The examples in this section demonstrate some basic patterns for asynchronous programming using Actors in Newspeak 4. We begin by trying to answer the question of how to translate synchronous code that uses immediate-sends to asynchronous code using eventual-sends.

5.5.1 Using Promise Pipelining

A transformation from synchronous code to asynchronous code always begins with substituting an eventual-send for an immediate-send in some method doX:

```
public doX: a <SomeClass> ^ <Object> = (
  | b | “local variables”
  b:: a doY. “setter send - equivalent to ‘b = a.doY();’ in Java”
  ^b doZ.
)
```

The code above contains a bit of new Newspeak syntax that deserves explanation. The vertical bars on the first line of the method enclose a space-separated list of local variables (in this case a single variable named `b`). Double-quoted strings contain comments, which are used to explain the Newspeak syntax in the above example. The `::` notation represents a setter-send, which (in this specific example) is functionally equivalent to an assignment operator in conventional languages.

Let’s assume we want to perform the `doY` operation asynchronously. As we saw in the factorial example, the first step is to replace the immediate-send of `doY` with an eventual-send.

```
public doX: a <SomeClass> ^ <Object> = (
  | b |
  b:: a <-: doY.
  ^b doZ. “error - promises can only accept eventual-sends”
)
```

Since the receiver of the `doZ` message is the result of the `doY` message send, we can no longer send `doZ` immediately. The solution is simply to replace the immediate-send of `doZ` with an eventual-send as well:

```
public doX: a <SomeClass> ^ <Promise[Object]> = (
  | b |
  b:: a <-: doY.
  ^b <-: doZ. “OK”
)
```

Note that we also changed the return type annotation of the `doX:` method from `Object` to `Promise[Object]`. Once a method is transformed to use asynchronous eventual-sends, if the return value of the method depends on any of the asynchronous operations, then the asynchrony creeps up to the method return value, which now becomes a promise (or a far reference) and the process of replacing immediate-sends with eventual-sends must proceed up along the caller hierarchy of `doX:`. In other words, methods that call `doX:` and make use of the

return value of doX: must also be modified to use eventual-sends when invoking doX:.

5.5.2 Using whenResolved:

Sometimes the code that follows a message send expects the actual value of the result to be available. This is the case when the result of a computation is used as the argument of a subsequent message send instead of as the receiver:

```
public doX: a ^ <Object> = (
  | b |
  “... preceding code ..."
  b:: a doY.
  “... subsequent code ..."
  ^doZ: b.
)
```

Assuming the doZ method expects the actual value of b, we can translate the above method to use asynchrony for doY as follows:

```
public doX: a ^ <Promise[Object]> = (
  | b |
  “... preceding code ..."
  b:: a <-: doY.
  ^b whenResolved: [
    “... subsequent code ..."
    doZ: b.
  ].
)
```

The gist of the transformation is wrapping the code that follows the eventual-send in a closure that is passed in a whenResolved: message to the promise b. If the wrapped code contains a return statement (as in this example) an additional step is required. Return statements must be removed from the wrapped code, and the return statement is moved up to the whenResolved: expression. The whenResolved: message on Promise makes it possible to obtain the return value of the subsequent code because whenResolved: returns a new promise for the value of the closure that is passed to it.

In the example above, there is a single return statement at the end of the subsequent code, which represents the simplest possible case. If there are multiple return statements, the wrapped code must be rewritten to save the desired return value explicitly and make that the value of the last expression of the closure. This kind of transformation is beyond the scope of this discussion.

5.5.3 Dealing with Loops

The above simple transformations do not apply when the code involves loops or other non-trivial control structures. In the case of loops, the standard way to transform the code is by first translating the loop to tail recursion. If there are multiple nested loops, this might also require refactoring nested loops out into separate methods. Here is an example of transforming a simple Read-Eval-Print loop (REPL):

```
public repl = (
```

```
  [ running ] whileTrue: [
    | line result |
    line:: console readLine.
    result:: self eval line.
    console printLine: result.
  ].
)
```

Using eventual-sends to access the console, the code would look like this:

```
public replAsync = (
  [ running ] ifTrue: [
    | line result |
    line:: console <-: readLine.
    line whenResolved: [
      result:: self eval line.
      console <-: printLine: result.
      replAsync. “tail-recursion”
    ]
  ].
)
```

Note that we had to use `whenResolved:`, since the `eval` message assumes that the actual contents of the line are available (`line` must be resolved when we call `eval`). Furthermore, note that since we do not expect a return value from the `console printLine:` message, the transformation of this message to an eventual-send did not require any additional steps besides inserting the `<-:` eventual-send operator.

5.5.4 Waiting for Several Results

Sometimes the result of several computations is necessary for the next step:

```
public combine = (
  | a b |
  a:: self calculateA.
  b:: self calculateB.
  process: a and: b.
)
```

If we want to compute `a` and `b` asynchronously, we can wait for both results by combining the two promises via the comma (`,`) operator:

```
public combine = (
  | a b |
  a:: self <-: calculateA.
  b:: self <-: calculateB.
  (a, b) whenResolved: [
    process: a and: b.
  ]
)
```

The above example is the simplest way to group and await multiple promises at once. The combined promise `(a, b)` forms a logical conjunction of `a` and `b` -- it will resolve only when both `a` and `b` have resolved, and will be smashed with an exception if either `a` or `b` gets smashed. The implementation of promise conjunction is based on the `whenAll` example described in [13].

5.5.5 MapReduce

A common pattern that occurs in asynchronous, concurrent processing is to launch a number of tasks and then combine their results. In the sequential case, the variable number of tasks can be processed one at a time and then the results combined:

```
public mapReduce: tasks = (
  | results <Collection[Object]> |
  “map tasks to results”
  results:: tasks collect: [ :task |
    task process.
  ].
  “... process (reduce) the results ..."
)
```

The asynchronous version can be written using the `PromiseGroup` class, which combines a variable number of promises into one promise for a sequence of all results:

```
public mapReduceAsync: tasks = (
  | resultPromises <Collection[Promise[Object]]>
    results <Promise[Collection[Object]]> |
  “map tasks to results”
  resultPromises:: tasks collect: [ :task |
    task <-: process
  ].
  results:: PromiseGroup for: resultPromises.
  results whenResolved: [
    “... process (reduce) the results ..."
  ]
)
```

Note that the `resultPromises` temporary variable is introduced in the place where the results variable was in the synchronous example. We cannot use `resultPromises` directly, because it contains a collection of promises. The simplest way to combine the promises into a single promise is to use the `PromiseGroup for:` constructor. The `,` (comma) operator on `Promise` that was used previously returns a `PromiseGroup` object, and the `PromiseGroup` class supports the `,` operator as well as the `whenResolved:` protocol of `Promise`.

5.6 Publisher/Subscriber

Implementing an asynchronous publisher/subscriber service in Newspeak 4 is trivial as illustrated in the following example implementation of a `StatusHolder` class, adapted from [13]:

```
class StatusHolder initialStatus: s = (| “slots”
  private status ::= s.
  private listeners = MutableArrayList new.
```

```
|) ( “instance methods go here”
  addListener: newListener = (
    listeners add: newListener.
    newListener <-: value: status.
  )
  setStatus: newStatus = (
    status:: newStatus.
    listeners do: [ :listener |
      listener <-: value: status.
    ].
  )
) : ( “class (static) methods go here”
)
```

A status holder object has two slots (fields), which store a status object, and a list of listeners to be notified when the status changes. The `addListener:` method appends the given listener object to the list of listeners and notifies the new listener of the current status. The notification to the new listener is sent asynchronously via the `value:` message -- by convention all Newspeak closure objects that accept a single argument support the `value:` message. The `setStatus:` method updates the status and notifies all registered listeners of the new value. Notification is again done asynchronously, and the `setStatus:` method returns immediately without waiting for listeners to process the notification. The `setStatus:` method can be invoked many times in a short period of time and the entire sequence of changes will be enqueued to all registered listeners for eventual processing at a later time and independently by each listener. Listeners can reside in the same actor, a different actor running in the same Newspeak VM, or a distributed actor running on a different node and communicating over a network[8].

In the above example, any object with a reference to the status holder can take on the role of either a publisher -- via the `setStatus:` method, or a subscriber -- by enlisting a listener via the `addListener:` method. The example can be extended by separating the authority to subscribe and publish into two nested class objects (similar to the example in [13]).

```
class StatusHolder2 initialStatus: s = (| “slots”
  private status ::= s.
  private listeners = MutableArrayList new.
|) ( “instance members”
  class Subscriber = (||) () : ( “class members”
    addListener: newListener = (
      listeners add: newListener.
      newListener <-: value: status.
    )
  )
  class Publisher = (||) () : ( “class members”
    setStatus: newStatus = (
      status:: newStatus.
      listeners do: [ :listener |
```

[8] Support for distributed actors is not implemented in Newspeak 4

```
          listener <-: value: status.
        ].
      )
    )
  ) : ()
```

Each instance of `StatusHolder2` contains its own `Subscriber` and `Publisher` classes. Objects with a reference to the `Subscriber` class object can only take on the subscriber role by enlisting a listener, and objects with a reference to the `Publisher` class object can only publish status changes.

5.7 Breadth-first Tree Walking Using Recursion

The following is a simple binary tree implementation in Newspeak with a method to walk the tree recursively in depth-first fashion:

```
class BinaryTree data: d left: l right: r = (|
  private data = d.
  private left = l.
  private right = r.
) (
  depthFirstWalk: handler <[]> = (
    handler value: self.
    left isNil ifFalse: [ left depthFirstWalk: handler ].
    right isNil ifFalse: [ right depthFirstWalk: handler ].
  )
) : ()
```

What if we wanted to add a breadth-first walk method? The textbook breadth-first search example uses a queue and a loop:

```
breadthFirstWalk: handler <[]> = (
  | queue = MutableArrayList new. |
  queue add: self.
  [ queue isEmpty ] whileFalse: [
    | node = queue removeFirst. |
    handler value: node.
    node left isNil ifFalse: [ queue add: node left ].
    node right isNil ifFalse: [ queue add: node right ].
  ].
)
```

In Newspeak 4, we can implement breadth-first walk in a simpler way via the use of asynchronous message passing and recursion:

```
breadthFirstWalk2: handler <[]> = (
  handler value: self.
  left isNil ifFalse: [ left <-: breadthFirstWalk: handler ].
  right isNil ifFalse: [ right <-: breadthFirstWalk: handler ].
```

```
)
```

The implementation above is almost identical to the `depthFirstWalk:` method, with only the recursive method calls changed to their asynchronous equivalent. This implementation is much simpler and easier to read, mostly because it does not require the explicit use of a queue data structure. Instead of explicitly using a queue, the recursive asynchronous message sends make use of the internal queue of the current actor. In this sense, asynchronous message sends act as an abstraction that hides a queue data structure, in the same way that traditional immediate message sends (the equivalent of a method or procedure call) hide a stack in their implementation. In fact, we could rewrite the `depthFirstWalk:` method to use a stack data structure in a loop, simply by modifying the `breadthFirstWalk:` method to send the `removeLast` message to `queue` instead of `removeFirst`, in effect utilizing the `queue` object (a generic `MutableArrayList` instance) as a stack.

5.8 Modules, Mutual Recursion and Actors

Top-level classes in Newspeak effectively function as modules [19]. The Newspeak language does not have a global namespace and all imports are passed as parameters to a top-level class constructor. A top-level Newspeak class is a fully self-contained parametric namespace. Instances of a top-level class can only access classes and objects given to it as parameters at construction time. Newspeak supports mutually recursive modules via the use of simultaneous slots. Simultaneous slots support mutual recursion via lazy initialization. The following example contains two module definitions -- `A` and `B`, which are instantiated in a mutually recursive fashion by a third module called `Whole`.

```
class Whole = (
|| “double-bars denote simultaneous slots”
  partA = PartA using: self.
  partB = PartB using: self.
||
)()

class PartA using: whole = (
  | Y = whole partB Y. |
)(
  class X = ()()
)
class PartB using: whole = (
  | X = whole partA X. |
)(
  class Y = ()()
)
```

In the example above, the `partA` and `partB` slots of `Whole` must be simultaneous (lazy), because the default factories (constructors) of both `PartA` and `PartB` refer to each other via their references to the instance of the `Whole` module object. This is a non-trivial example of indirect mutual recursion that poses challenges for the implementation of lazy slots in Newspeak

[20]. Implementing the same type of mutually-recursive structure with lazy vals in scala, for example, requires that the X and Y slots of PartA and PartB be lazy as well in order to avoid divergence [21].

Actors in Newspeak 4 can be used to provide an alternative solution for mutually recursive modules. We can create a new version of the Whole module, which creates PartA and PartB as actors:

```
class WholeActors usingActors: actors = (
|
  partA = (actors createActor: PartA) <-: using: self.
  partB = (actors createActor: PartB) <-: using: self.
|
)()
```

The createActor: method of the built-in actors module creates a new actor and seeds it with an instance of the given class object. We then invoke the factory (constructor) of the class asynchronously. Note that in this case the partA and partB slots do not need to be simultaneous. Since partA and partB are now actors they must communicate with each other, and with WholeActors exclusively via asynchronous message sends. The factories of PartA and PartB must be rewritten to obtain each other's reference asynchronously as well:

```
class PartA using: whole = (
  | Y = whole <-: partB <-: Y. |
...
class PartB using: whole = (
  | X = whole <-: partA <-: X. |
...
```

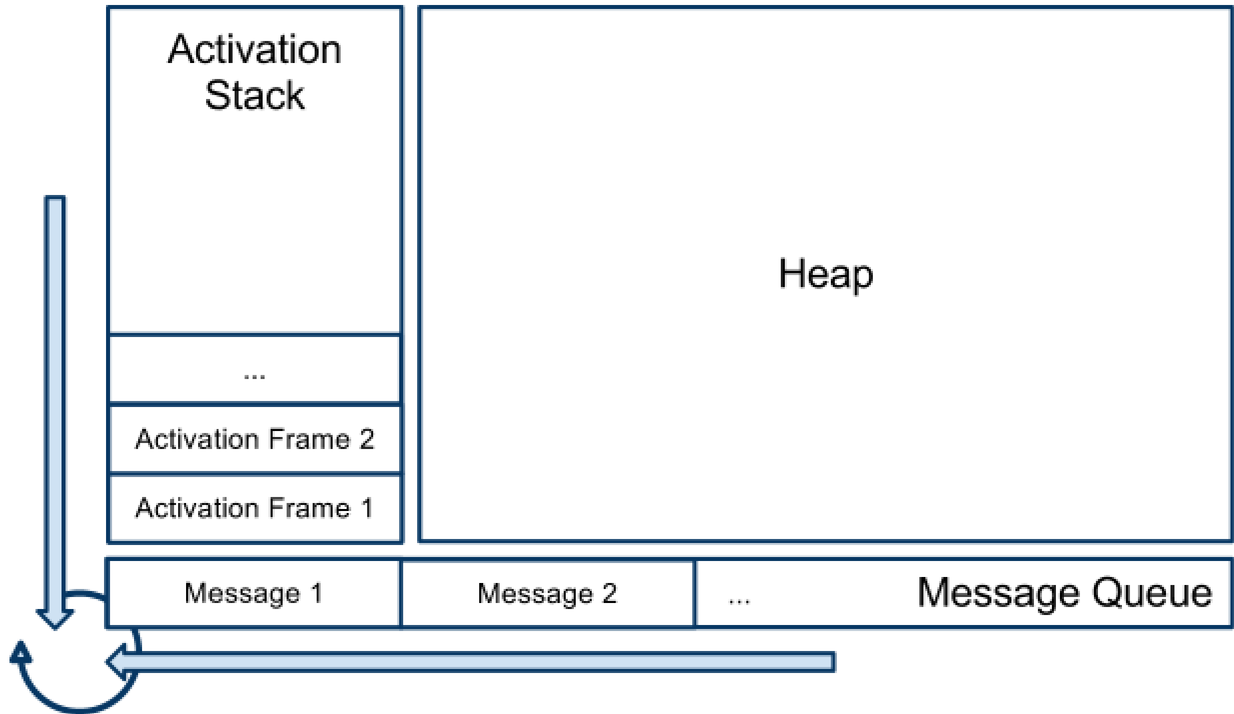
The above example illustrates how actors naturally support mutual recursion. Asynchronous communication via eventual-sends in Newspeak is, in effect, lazy.

6 Functional Specifications

6.1 Introduction

A Newspeak **actor** is a virtual process (equivalent to the **vat** in E [13]) and consists of the following parts:

- Activation Stack
- Message Queue
- Heap



An actor is similar in structure to an Operating System process or thread, with the exception of the addition of a message queue. The message queue is managed by the actor and continuously serves messages, which are then pushed onto the activation stack and executed as conventional method calls. The execution of an actor proceeds in a right-to-left and top-to-bottom order.

6.2 Messages

Objects in a actor's heap can exchange messages with each other via immediate- or eventual-sends.

6.2.1 Immediate-send

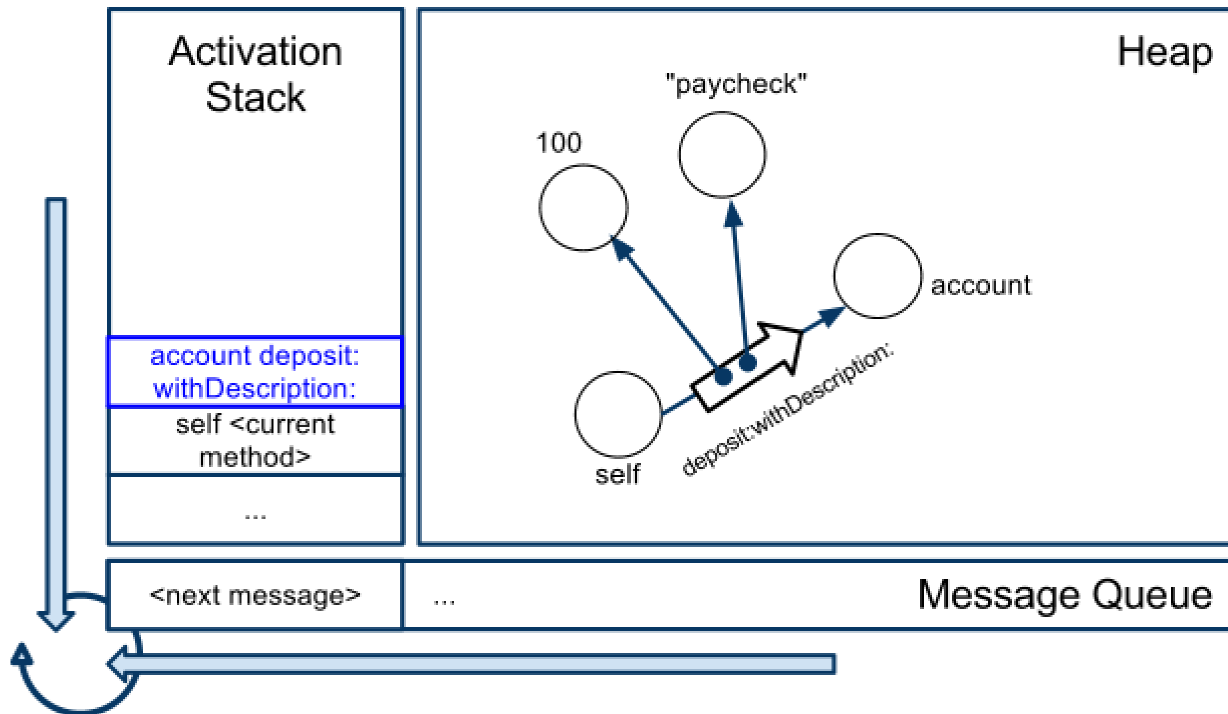
An **immediate-send** is the equivalent of a standard method call in languages like Java, or a message send in Smalltalk. Newspeak immediate-sends build on and extend the Smalltalk message send syntax:

```
account deposit: 100 withDescription: ‘paycheck’.
```

The above code corresponds to

```
account.depositWithDescription(100, “paycheck”);
```

in traditional Java method call syntax. An **immediate-send** pushes a new activation frame on the **activation stack** with the message selector, arguments and return address, and proceeds to execute the method immediately.



The heap section of the above picture contains a Granovetter diagram [13] representing the immediate-send of the `deposit:withDescription:` message to the `account` object. Each circle represents an object, each arrow represents a reference, and the boxed arrow represents the immediate-send of the `deposit:withDescription:` message from the object of the currently-executing method (`self`) to the `account` object. The arrows coming out of the boxed arrow represent the object references carried as arguments inside the message. The new activation frame created as a result of the immediate-send is shown at the top of the stack in blue. An immediate-send does not change the message queue.

6.2.2 Eventual-send

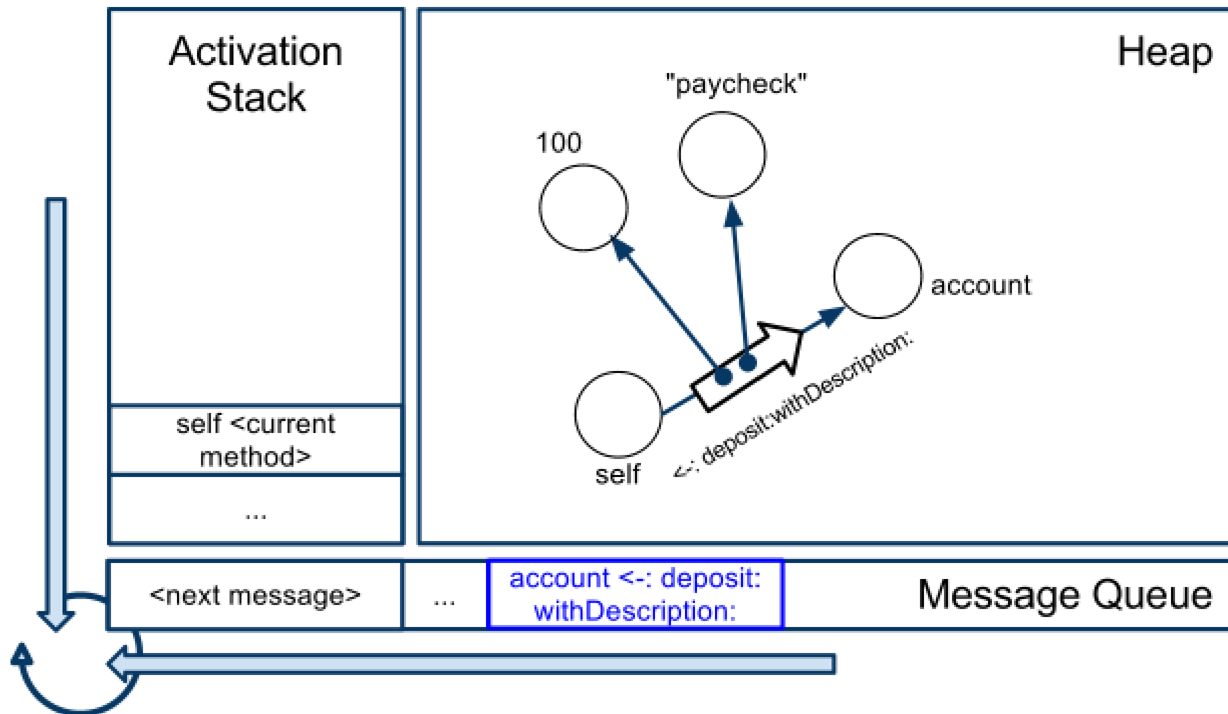
In addition to the standard, familiar behavior of immediate-sends, objects inside an actor can also communicate via eventual-sends, courtesy of the message queue. An **eventual-send** enqueues a message onto the **message queue** to be executed at a later time and returns immediately to the caller. Our immediate-send example from the previous section can be converted to an eventual-send as follows:

```
account <-: deposit: 100 withDescription: ‘paycheck’.
```

The above operation corresponds to

```
account <- depositWithDescription(100, “paycheck”);
```

in the E and AmbientTalk languages, which follow the Java syntax for immediate-sends.



Instead of pushing the message on the stack and immediately executing it, the eventual-send operation adds the message to the end the message queue and immediately resumes execution of the current method. The message will be processed only after the current message has finished executing and all other messages between the current and the new message have been processed. The new message created as a result of the eventual-send is shown at the right end of the queue in blue. An eventual-send does not change the activation stack.

6.3 Activation Stack

The activation stack of an actor consists of **activation frames** just like the thread stack in traditional programming languages and operating systems. The activation stack is affected by immediate-send operations as described previously -- an immediate-send pushes a new activation frame on the stack and the activation frame is popped off the stack when the immediate-send returns.

Each activation frame consists of:

1. Receiver - the object receiving the message
2. Selector - the message selector, or method name
3. Arguments - a list of arguments passed to the method
4. Return address - the location, at which to resume execution after completion.

Local variable slots are also a part of the activation frame, but since those are allocated dynamically by the message receiver, we do not list them here.

6.4 Message Queue

The message queue holds a sequence of messages eventually-sent to the actor. Messages are processed in FIFO order. Newly arriving messages are appended to the tail of the queue and the next message to be processed is picked up from the head of the queue.

Each message in the queue consists of:

1. Receiver - a reference to the object receiving the message
2. Selector - the message selector, or method name
3. Arguments - a list of arguments passed to the method
4. Resolver - a message resolver

The structure of a message is almost identical to that of an activation frame -- it only lacks local slots, and instead of a return address it has a resolver. The resolver is an object, which is capable of transferring the result of the message processing back to the sender[9]. Unlike the return address of an activation frame, the resolver of a message is optional and can be missing in certain special cases.

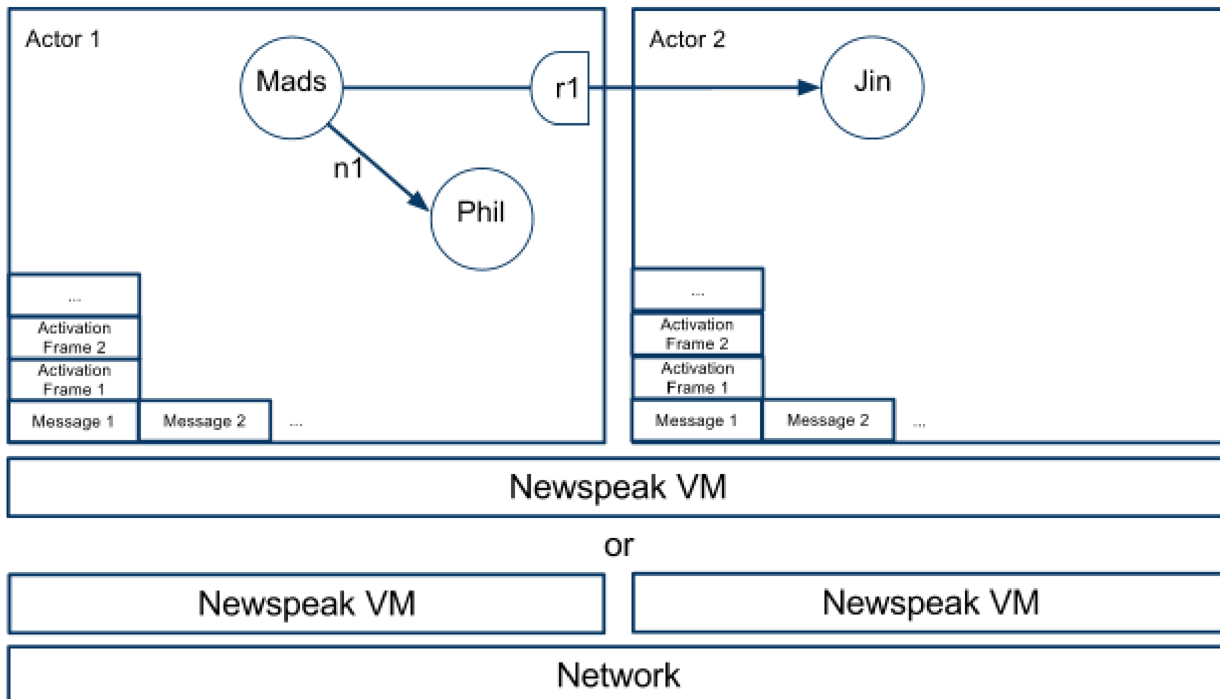
6.5 Heap

The heap of an actor is where all objects are allocated. Heaps of different actors are virtually isolated, and each actor only has access to its own heap. The heaps are only virtually isolated, because in practice all actors running within a single shared-memory Newspeak VM share that VM's heap. Virtual isolation is achieved by tightly controlling the flow of data and references between actors -- transitively immutable (value) objects are passed by value, and references to transitively mutable objects are encapsulated in far references, which can only accept eventual sends that get posted on the message queue of the target object's actor.

6.6 Actor Communication

Multiple actors can reside within the same Newspeak VM. Objects in one actor can hold references to objects residing in other actors. Such object references are called **far references**. Far references are one kind of **eventual reference**. Promises, which are discussed later, are another kind of eventual reference. Eventual references are defined by the fact that they only accept eventual-sends. Immediate-sends on eventual references are not allowed.

[9] Resolvers and promises are fully explained in Section 6.15.



In the picture above, the object Mads resides in Actor 1 and holds a far reference `r1` to object Jin, which resides in Actor 2. The far reference `r1` is a proxy, which accepts eventual-sends and forwards them to Jin by placing them on Actor 2's message queue. Attempting to immediate-send a message to `r1` results in an error.

Mads also holds a direct reference `n1` to Phil, who resides in the same actor. Regular direct object references are called **near references** in order to distinguish them from eventual references. Near references such as `n1` can accept both immediate- and eventual-sends.

The actors in the above diagram can live within the same Newspeak VM, or on different Newspeak VMs communicating over a network. Unless otherwise noted, all further discussions on semantics of inter-actor communication apply to both the single-VM and multi-VM case. Multi-VM (aka distributed) actors are not implemented in Newspeak 4, however, their behavior is fully specified in this chapter, in order to ensure that the implementation is future-proof and can be extended to support the multi-VM case as transparently as possible.

6.7 Reflection and Actors

6.6.1 Introduction to Mirrors

Reflection in Newspeak is provided via mirrors [23]. In contrast to languages like Java or C# where the reflection API is embedded in the Object class protocol and therefore publicly visible to all running code, mirrors encapsulate reflection capabilities in external adapter objects, which enables the separation of different reflection capabilities and the restriction of access to reflection via capability-based security -- an object can only access reflection capabilities that are explicitly exposed to it via references to mirror library objects that provide reflection services.

The actor mirror library in Newspeak 4 provides mirrors for reflecting on references, actors, and actor messages.

6.6.3 The Reference Mirror

A reference mirror reflects an object reference and can be used to determine whether the reference is near or far and to extract the referent from far references. In addition, the reference mirror provides access the actor mirror for the actor of the referent.

6.6.2 The Actor Mirror

Actor mirrors can be retrieved based on an eventual reference of the currently executing actor. An actor mirror provides access to the activation stack and message queue of an actor.

6.6.3 Eventual-send Proxies

Eventual-send proxies are proxy objects, which accept immediate-send messages and convert them into eventual-sends to the target reference. For example, the following two lines of code produce equivalent results:

```
joe <-: hello.
(actorMirrors forReference: joe) esendProxy hello.
```

Both lines result in an eventual-send of the `hello` message to `joe`, even though the second message is syntactically an immediate-send to an esend-proxy object for `joe`.

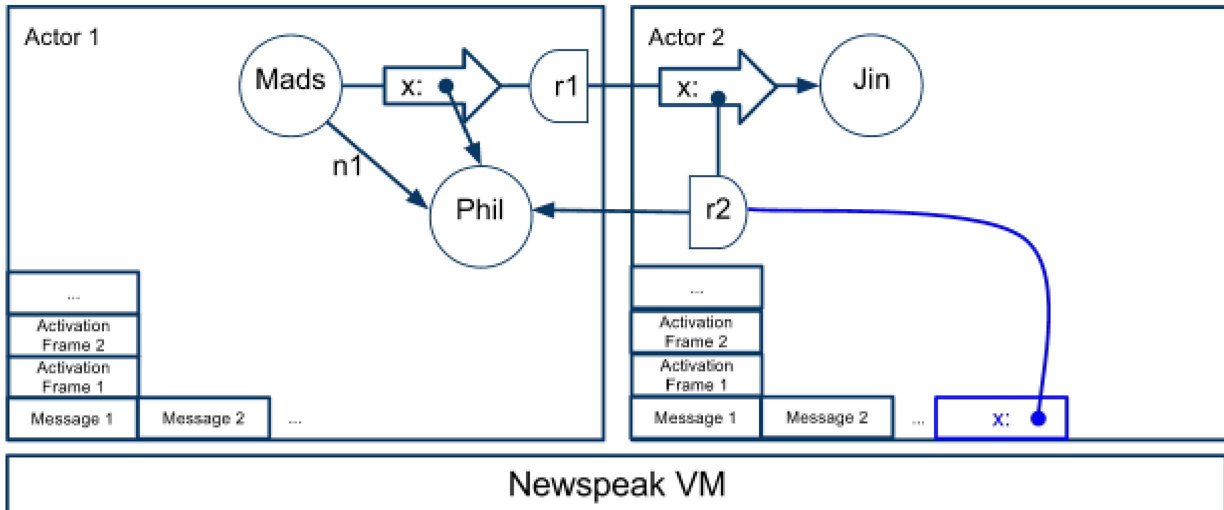
6.8 Argument-passing Semantics

Newspeak supports the notion of a value object (as defined in Section 3.1.1 of the Newspeak Language Specification [2]). A value object is an object whose contents are deeply immutable and whose identity is based on its contents (multiple instances of a value object with the same contents are virtually indistinguishable from each other). In eventual-sends across actors, value objects are passed by value, and all other objects are passed by far reference. The same semantics apply to both the arguments and the return value of a message.

6.8.1 Pass-by-Far-Reference

In the following diagram, Phil is a mutable object, and Mads passes a reference to Phil in message `x:` to Jin (double quotes are comment delimiters in Newspeak):

```
“Mads says:”
r1 <-: x: n1 “Phil”.
```

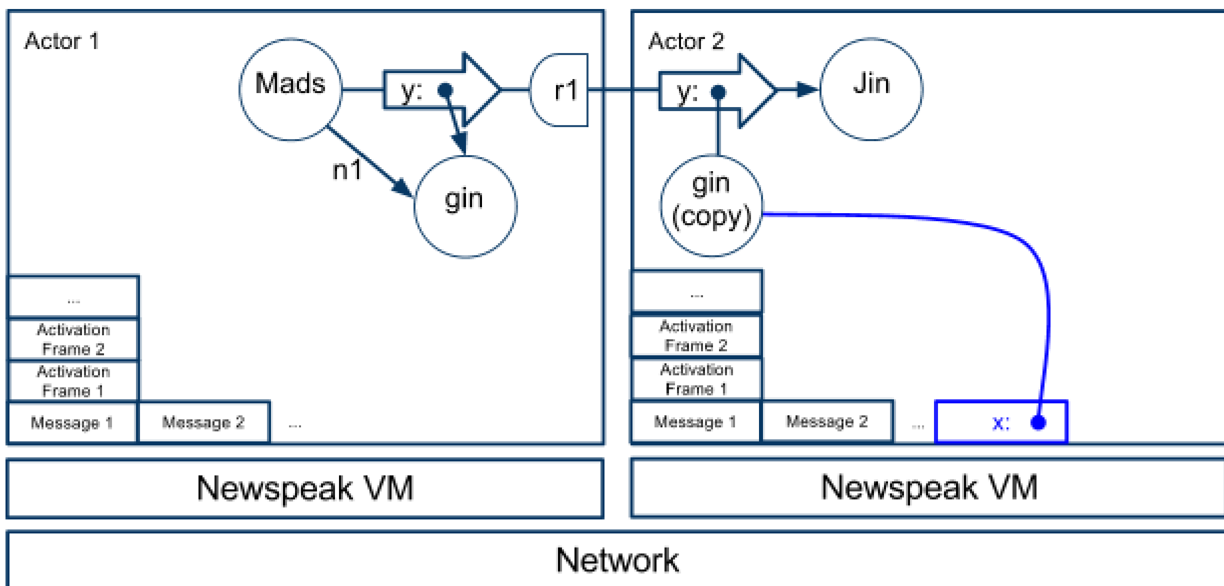


As the message `x:` Phil travels across actor boundaries from the far reference `r1` to the actual receiver Jin, a new far reference `r2` is created inside Actor 2 pointing back to Phil in Actor 1. The message `x:` Phil is also shown in blue as the new last message on the queue of Actor 2, holding a far reference to Phil. The message also holds a near reference to its receiver Jin, and a reference to the resolver of a promise given to Actor 1, even though those references are not shown in the diagram.

6.8.2 Pass-by-Value

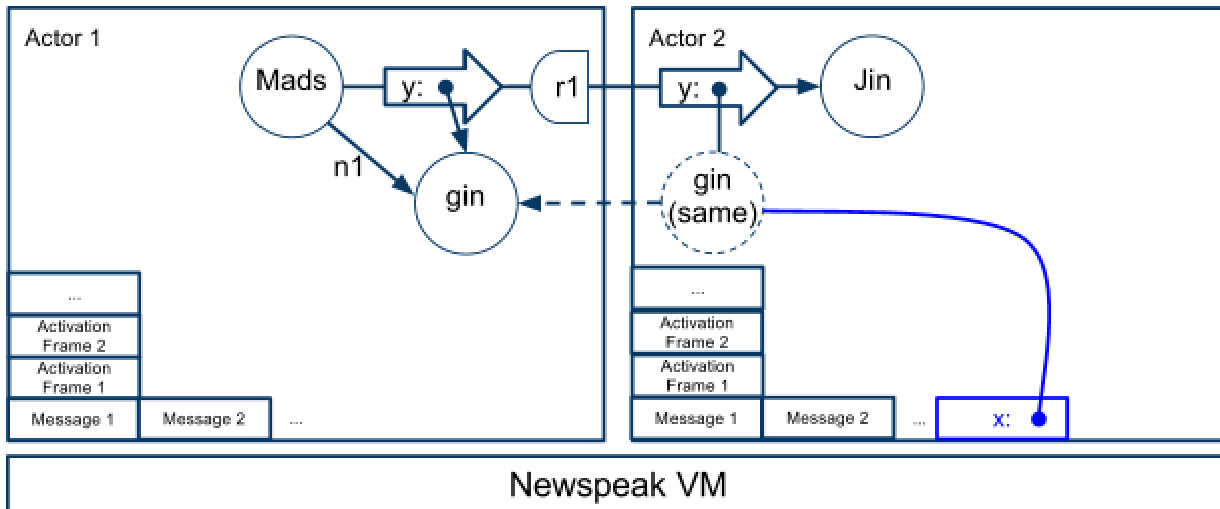
In the following diagram, gin is a transitively immutable value object (of questionable content), which is passed as an argument in a message from Mads to Jin:

```
"Mads says:"
r1 <-: y: n1 "gin".
```



In the multi-VM case, as the message `y:` gin passes across from Actor 1 to Actor 2, a deep copy of the contents of object gin is serialized from Actor 1's heap, transferred over the network, and deserialized into Actor 2's heap. The new message in Actor 2's message queue then holds

a reference to Actor 2's copy of the gin object.



When Actors 1 and 2 both reside in the same Newspeak VM, the serialization and deserialization steps steps are omitted for performance reasons. The new message in Actor 2's queue holds a direct reference to the object gin inside Actor 1's heap. Since gin is a transitively immutable value object, references to it can be safely shared between actors, while still maintaining the virtual isolation of actors' heaps. Thus, within a single Newspeak VM only an actor's mutable objects are truly isolated, via far reference encapsulation.

6.8.3 Far Reference Passing Semantics

An individual actor cannot contain far references between objects in its own heap. This is a conscious design choice aimed at keeping the semantics and functionality of actors simple and predictable. To enforce this rule, far references to objects in a receiver's heap are unboxed to near references as part of message dispatch .This is shown in the following example.

- Actor A holds a reference `r1` to object X in actor B's heap.
- Actor A sends `r1` in a message to actor B.
- `r1` is unboxed and actor B now holds `r1'` -- a near reference to object X in its own heap.

If an actor could have both a far and a near reference to an object in its heap, then comparing those references becomes problematic. From the point of view of a programmer, it is simpler and more consistent if all objects within the same actor's heap can immediately reach each other.

Since value objects are always passed by-value, far references can only point to non-value objects in other actors' heaps. Far references like `r1` are also not treated as value objects themselves. This means that a value object cannot contain far references and therefore message passing between actors running in the same address space (single-VM case) does not require deep serialization/deserialization of objects.

Furthermore, the above semantics imply that eventual-sends, where both sender and receiver reside inside the same actor, follow the same semantics as those for immediate-sends -- all objects are passed by near reference.

6.9 The Glue Holding Actors Together

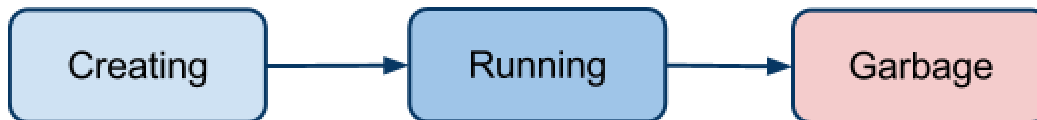
Eventual references are the glue that binds actors together. The only way actors can affect each other (besides the special case of creating a new actor) is by eventual-sending messages to one another. In Newspeak, the only way actors can send messages to each other is via eventual references. In the fundamental actor model, as described by [11], actor messages can be arbitrary tuples of objects. In contrast, actor messages in Newspeak follow a stricter form, as every message is the 4-tuple described earlier, consisting of receiver, selector, arguments and resolver. An eventual reference is a reference to an object across actor boundaries. In other words, an eventual reference is a reference that resides in one actor's heap but points to an object in another actor's heap. Eventual references take their name from the fact that they can only accept eventual-sends.

6.10 Eventual-sends

As in Smalltalk, objects in Newspeak communicate solely by exchanging messages. Messages can be sent in two ways -- immediate-send and eventual-send. An immediate-send is the equivalent of a method call in standard object-oriented terminology. It has the same constituent parts -- a receiver (the object on which the method is called), a selector (the method name), and a list of arguments. An immediate-send creates a new activation frame on the actor's activation stack and proceeds to immediately execute the selected method in the context of that new activation frame, just as a method call does.

An eventual-send functions in an analogous way, with the primary difference that instead of immediately executing the selected method, it enqueues a message on the actor's message queue.

6.11 Actor Lifecycle



An actor can be in one of three states -- creating, running and garbage. Creating is an ephemeral state and only included here for completeness. User code never sees actors in the creating state -- the actor framework is free to complete the creation of the actor either before or after (asynchronously) returning a reference to the application. There is also no created state and no explicit operation to "start" an actor. An actor enters the running state as soon as it is created and is immediately ready to receive and process messages. If an actor is created asynchronously, then actor creation will return a promise for the result of the actor creation. Since actor creation always results in a far reference, this is transparent to the user of the actor framework. Far references and promises are both eventual references, and in this sense they are indistinguishable from the point of view of the user.

While in the running state, the execution of each actor proceeds in an infinite loop around the message queue. At every iteration of the loop, the actor completes one turn of its execution. The actor takes a message from the head of the queue, translates it into an immediate-send to the message's receiver and thus converts the message into an activation frame. Once this immediate-send completes, the result is sent to the resolver, or if an exception is thrown, that exception is forwarded to the resolver instead.

An actor's existence comes to an end when the actor becomes garbage. An empty queue

does not lead to the termination of an actor -- if the actor's queue is empty, that actor becomes quiescent until a new message arrives on the queue, as long as there are far references pointing to objects in the actor's heap. An actor becomes garbage once its queue is exhausted and there are no far references pointing to objects in its heap. At this point, the objects in the actor's heap, including its queue are garbage collected just like any other objects in the Newspeak VM. In effect, an actor does not have a lifetime of its own, instead its lifetime is that of all objects on its activation stack, message queue and heap.

6.12 Actor Creation

Actors are created via an actor factory provided by the standard Newspeak platform. An actor is brought into being by bootstrapping its heap with a single instance of a class object[10] -- this initial class object is special and is called the actor's **primordial class**. The primordial class is provided in the form of a mixin[11] of a top-level class. This ensures the complete virtual separation of the heaps of actors running on the same Newspeak VM.

The result of creating an actor is an eventual reference to the actor's primordial class object. The new actor's message queue is not automatically seeded with any messages. To make the new actor do something, the current actor sends eventual messages to the new actor's primordial class object. Usually a newly created actor is initialized by sending the primordial class its factory message, which creates an object in the new actor's heap and returns a far reference to it.

Creating an actor happens in three steps. First, the actor's heap is seeded with a copy of the supplied mixin object[12]. Second, the mixin is applied to Object in order to produce the primordial class object. Third and last, the actor is enabled for execution and enters the running state. The first step is only a conceptual one because in practice the mixin object is a value and can be reused. Although the second step must conceptually occur in the context of the newly created actor, it is executed immediately from the current actor's context because it is a known non-blocking operation.

The above applies to actors created inside the same Newspeak VM. Remote actors can only be created by eventual-sends to objects living inside the remote VM, which indirectly expose their platform's actor factory.

6.13 Bootstrapping Newspeak

Execution of a Newspeak VM begins with the creation of a new actor -- the **main actor**. The main actor has a primordial class object just like any other actor in Newspeak. That main class is instantiated and a main: message is eventual-sent to the new instance. This begins the execution of the new Newspeak VM.

Since the main actor is created outside of the context of another actor, the process of its creation is handled by bootstrapping code and the above steps execute in sequence without the use of eventual-sends and promises.

---

[10] Classes in Newspeak are first-class objects that act as a factory for their instances.

[11] Mixins in Newspeak are immutable objects, which capture a class declaration's code and nothing else.

[12] Mixin application in Newspeak results in the construction of a class object -- in this case the actor's primordial class, which is the first object created in the heap of the new actor.

6.14 Actor Termination

There is no explicit operation to terminate an actor, because such an operation is not needed. While actors require external resources -- OS threads or processes, in order to execute, these execution resources are not statically allocated to individual actors. Instead, the actor framework internally manages the available execution resources, automatically allocating and releasing them, and scheduling actors among them dynamically, in the most efficient manner that the framework can.

An actor becomes garbage when the actor's queue is exhausted, the actor has completed its last turn, and no far references remain to objects in the actor's heap. The second condition is important. It means that the actor's message queue lifetime is tied to that of the actor's heap. This is a natural dependence -- in practice if an actor's queue is reified as a distinct object by the actor framework implementation, far references will necessarily hold (possibly indirect) references to this object. This is not prescriptive, however -- a simple actor framework can schedule the messages of many or all actors on a single queue, in which case actor creation and disposal does not involve the explicit management of per-actor queue objects. A more complicated actor framework implementation might maintain individual message queues for each actor, so that it can efficiently utilize available execution resources by scheduling actors across multiple hardware threads, cores and processors. In this case, separate lists of empty and non-empty actor message queues might have to be maintained. These two lists are analogous to the running and blocked queues typical of OS thread schedulers. The empty message queue list (the blocked queue) must hold weak references to actors' queues, in order to allow these message queues to be garbage collected in case no far references to the corresponding actors remain.

6.15 Futures in Newspeak

Up to this point, the discussion of actor communication (via eventual-sends) only focused on passing parameters as part of a message to an actor. An eventual-sent message can also result in a response (return value) being sent the message's sender. Return values in eventual-sent messages are passed via **futures**. An Actor Future in Newspeak consists of two distinct facets -- a promise and a resolver. As part of the construction of an eventual message, a future is created and the resolver of that future is transmitted along with the message arguments to the message recipient, while the promise facet of the future is returned immediately to the message sender. The recipient of the message uses the future resolver to send the return value back to the message sender and the sender in turn can use the promise to register a listener to be notified when the return value becomes available.

Promises are a type of eventual reference, meaning they only accept eventual-sends. The other type of eventual reference -- far references, were discussed earlier as part of parameter-passing semantics.

6.15.1 Promise Listeners

The following sample code demonstrates the use of a promise to register a listener for the return value of an eventual-sent computation:

```
p:: math <-: factorial: 10.
p whenResolved: [
```

```
  transcript show: ‘factorial of 10 = ‘ + p.
]
```

The first line sends a message to the `math` object asking it to compute the factorial of `10` and stores the promise for the result in `p`. On the second line a block closure object is registered with the promise to execute when the result of the computation becomes available. The closure block executes in a later turn of the current actor. At that time the promise `p` holds the result of the computation and will forward all immediate- and eventual-sent messages to that result. Since the result is a number -- an immutable value object, `p` is now effectively a near reference to a `Number` object. Return values are passed between actors following the same semantics as those for parameter passing -- transitively immutable value objects are passed by value, and all other objects are passed by far reference.

6.15.2 Exceptions

A promise listener can also include a catch block, which will be executed if the processing of the message in the target actor completes abnormally with an exception:

```
p:: math <-: factorial: 10.
p whenResolved: [
  transcript show: ‘factorial of 10 = ‘ + p.
] catch: [ :e |
  transcript show: ‘exception occurred: ‘ , e.
]
```

If the processing of the `factorial` message results in an exception, the second block above will be executed with the exception passed in the `e` argument. Since exceptions are by convention value objects in Newspeak, `e` will hold a near reference to a copy of the exception triggered in the actor processing the factorial message.

6.15.3 Promise Sequencing

Multiple promises can be combined via the `,` (comma) operator, such that a block of code will execute only when all of the promises have been resolved:

```
p1:: math <-: factorial: 10.
p2:: math <-: factorial: 20.
(p1, p2) whenResolved: [
  transcript show: ‘factorial of 10, 20 = ‘ + p1 + ‘, ‘ + p2.
] catch: [ :e |
  transcript show: ‘exception occurred: ‘ , e.
]
```

The code above is semantically equivalent to the following:

```
p1:: a <-: factorial: 10.
p2:: a <-: factorial: 20.
p1 whenResolved: [
  p2 whenResolved: [
```

```
    transcript show: ‘factorial of 10, 20 = ‘ + p1 + ‘, ‘ + p2.
  ] catch: [ :e |
    transcript show: ‘exception occurred: ‘ , e.
  ]
] catch: [ :e |
  transcript show: ‘exception occurred: ‘ , e.
]
```

Notice that with promise chaining the exception block does not need to be replicated, and execution can be more efficient since the two promises will be awaited concurrently.

6.15.4 Promise Pipelining

Being eventual references, promises can receive eventual-sent messages. Before a promise is resolved, the target of eventual-sent messages is unknown, and so these messages are queued up inside the promise and forwarded to the target object when the promise is resolved. Messages eventual-sent to an already resolved promise are immediately forwarded to the target object (the resolution of the promise). The following example demonstrates the use of a promise before its resolution:

```
employee:: db <-: findEmployeeById: 100.
employee <-: promote.
```

First the db object is queried for an employee with id 100 using an eventual-send. The eventual-send enqueues the query with the database actor and returns immediately with a promise for the result of the query. The `promote` message is then saved by the promise and will be delivered to the actual employee object once the query completes and the result is available at some later point in time.

The result of an eventual-send to a promise is another promise, which is tied to the original promise. For example:

```
employee:: db <-: findEmployeeById: 100.
manager:: employee <-: getManager.
manager <-: promote.
```

In this case, when the code above executes both the `employee <-: getManager` and `manager <-: promote` messages are queued up for later delivery. Once the `db <-: findEmployeeById:` message is processed the employee future will be resolved, and it will eventual-send the `getManager` message to the actual employee object. Then once the `employee <-: getManager` message is processed, the `manager` future will be resolved and it will eventual-send the `promote` message to the actual manager object. This form of delayed execution is called promise pipelining [13].

6.15.5 Promises and Double Dispatch

Like in Smalltalk, primitive messages in Newspeak, such as addition (+), subtraction (-), multiplication (*) and division (/) are double-dispatched [24]. This means that if an argument to one of these messages cannot be coerced to a number of the same type as the receiver of the message, the implementation will attempt to send the message to the argument before giving

up. In the following example:

```
42 + n
```

the message + is sent to the object 42 with an argument n. If n is not an integer object, the + method will attempt to delegate the addition operation to n (double dispatch). One way to implement this is by sending the sumFromInteger: message to n [25]:

```
n sumFromInteger: 42
```

If n points to an unresolved promise the above message will fail, because unresolved promises do not accept immediate-sends. Given a means to determine if a method was invoked via an eventual-send, it is conceivable that double-dispatched methods can be adapted to support unresolved promises as arguments, by ensuring that the double-dispatch occurs in the same manner (eventual vs immediate) as the manner in which the current message was received. It is possible for the activation mirror protocol in Newspeak to be extended with information about the type of send that produced the current activation. However, this information is not presently available in Newspeak activation mirrors.

6.15.6 The Promise Protocol

The Promise protocol includes a small set of messages, such as the comma composition operator and the whenResolved:catch: message. In Newspeak these messages serve the role of the corresponding control structures, supported as part of the language syntax in E and AmbientTalk. In most cases, the Promise protocol does not interfere with promise pipelining (where the promise functions as a transparent proxy for its resolution) because pipelined message are always eventual, while messages sent directly to the promise are immediate.

In the example below, the resolution of p is an object that supports the whenResolved: message:

```
“wait for p to resolve”
p whenResolved: [ ... ].
“forward the whenResolved: message to p’s resolution”
p <-: whenResolved: ...
```

The first statement uses the Promise protocol to register a listener for the promise’s result, while the second message buffers the whenResolved: message to be delivered to p’s resolution at a later time. The two messages have the same selector, but produce different results and do not interfere with each other.

There is one case, which could be a source of confusion. In the example below, while the otherMessage: is forwarded to p’s resolution, the whenResolved: message is sent to the Promise p itself:

```
p whenResolved: [
      p whenResolved: ... “sent to the Promise p”
      p otherMessage: ... “forwarded to p’s resolution”
]
```

The solution in this case is to introduce a parameter to the block closure, which will be populated with the actual resolution of the promise:

```
p whenResolved: [ :r |
      r whenResolved: ... “sent directly to p’s resolution r”
      r otherMessage: ... “sent directly to p’s resolution r”
]
```

6.16 Order of Message Delivery

The actor model of computation [9] does not include any guarantees for the order of message delivery. While asynchronous messages sent between actors are guaranteed eventual delivery, the delivery can occur in any order, regardless of the order, in which messages were sent. While this makes the model very simple and flexible, in practice it imposes a heavy burden on today’s programmer who is accustomed to a more orderly and predictable world.

There are two scenarios in particular where ordering of messages can be useful, and a programmer might intuitively expect a certain order of message delivery.

The first scenario directly follows from the sequential nature of program execution in conventional programming languages. Given the following code where two messages are eventual-sent to the same actor `bob` one after the other:

```
bob <-: hello.
bob <-: world.
```

It seems natural to expect that `bob` will receive the `hello` message before the `world` message.

The second scenario involves the introduction of two actors to each other. Let’s assume that Alice says:

```
bob <-: initialize.
carol <-: sayHelloTo: bob.
```

And then in response to the `sayHelloTo:` message, Carol says:

```
sayHelloTo: friend = (
  friend <-: hello.
)
```

It seems natural to expect that Bob will receive the `initialize` message from Alice before the `hello` message from Carol. This guarantees that Bob will be initialized and ready to respond when he receives the `hello` message from Carol.

The above two ordering guarantees are exactly what is provided by E-ORDER [13], an order defined and used by the E language. E-ORDER can be summarized as follows:

1. All messages eventual-sent to the same reference from the same actor are delivered in the order, in which they were sent.
2. All messages eventual-sent to reference X from actor A before X is forwarded by A in message M to actor B are delivered before any of the messages sent from B to the

reference X that it received as part of message M,
Eventual-sent messages in Newspeak are delivered in E-ORDER.

6.16.1 Non-determinism in E-ORDER

E-ORDER is weaker than TOTAL ORDER and should not be confused with the latter. This section highlights what is **not** guaranteed by E-ORDER.

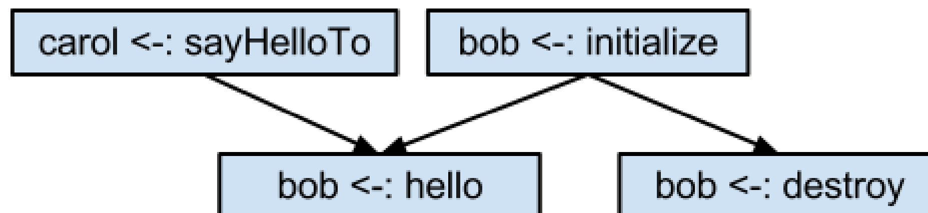
Expanding on the example from the previous section, we show several scenarios of non-determinism in message delivery that could be unexpected and ways to introduce determinism to match expected behavior by using the guarantees provided by E-ORDER.

6.16.1.1 Scenario 1

If Alice says:

```
bob <-: initialize.
carol <-: sayHelloTo: bob.
bob <-: destroy.
```

Bob might be destroyed before Carol gets a chance to say hello to him. The execution flow graph in this case looks like this:

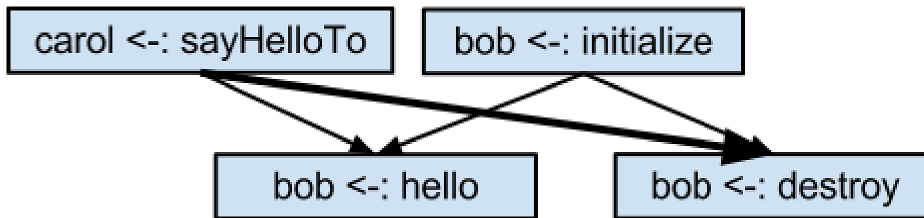


In the flow graph, each box represents a message and arrows indicate a **happens-before** relationship between messages. For example, `bob <-: initialize` is guaranteed to happen before `bob <-: destroy`. Notice that Carol might actually receive the `sayHelloTo:` message before Bob receives the `initialize` message, but Bob is still guaranteed to receive the `initialize` and `hello` messages in order, because of the second rule of E-ORDER. The first rule of E-ORDER dictates that Bob will receive the `destroy` message after the `initialize` message, but the `hello` and `destroy` messages can be delivered in any order relative to each other.

A first attempt to fix this might look like this:

```
bob <-: initialize.
(carol <-: sayHelloTo: bob) whenResolved: [
  bob <-: destroy.
]
```

However, this is still incorrect. In this case, Bob will be destroyed only after Carol is done processing the `sayHelloTo:` message but Bob might still be destroyed before he receives the `hello` message. As the updated execution flow graph shows, all we did was add a happens-before relationship between `carol <-: sayHelloTo:` and `bob <-: destroy`.



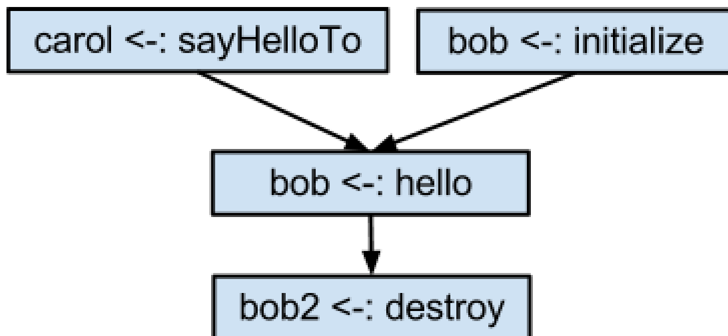
A proper way to fix this requires that we change the way Carol processes the `sayHelloTo:` message. Carol needs to send back the reference to Bob that she used so that we can use the second rule of E-ORDER to guarantee the order of delivery of the three messages to Bob:

```
sayHelloTo: friend = (
  friend <-: hello.
  ^friend.
)
```

Then Alice can say:

```
bob <-: initialize.
bob2:: carol <-: sayHelloTo: bob.
bob2 <-: destroy.
```

In this case, both `bob` and `bob2` are far references to the same target object -- Bob, but they are different in one very important way. Messages sent to `bob2` are guaranteed to be delivered after the `hello` message sent by Carol. This is illustrated in the updated execution flow diagram below:



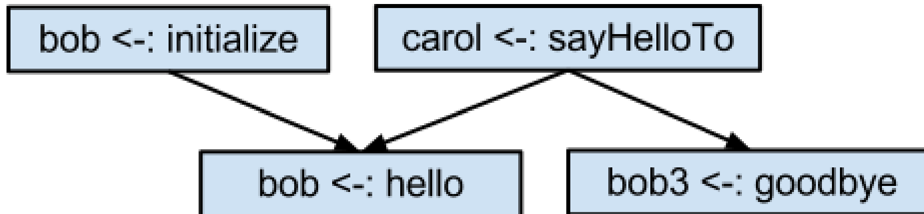
Note one very important difference in the diagram above from the previous diagrams: the `initialize` message no longer has a direct happens-before relationship to the `destroy` message. However, since `initialize` happens-before `hello` and `hello` happens-before `destroy`, by transitivity we can deduce that `initialize` happens-before `destroy`.

6.16.1.2 Scenario 2

If Carol already contains another reference to Bob -- `bob3`, which was obtained prior to receiving the message `sayHelloTo: bob`, then in the following example:

```
sayHelloTo: friend = (
  friend <-: hello.
  bob3 <-: goodbye.
)
```

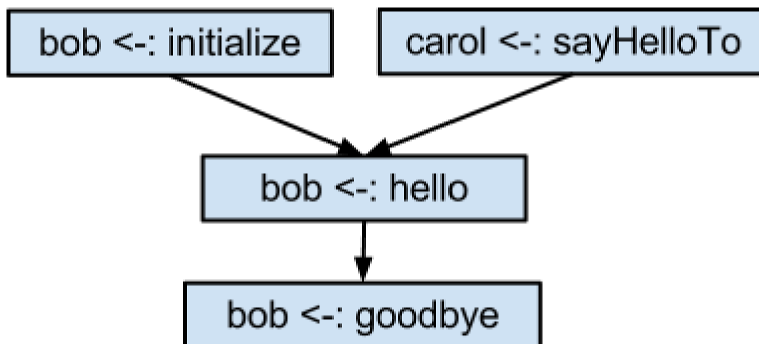
it is possible that Bob receives the goodbye message prior to the hello message. Note that in the code above both friend and bob3 are both references to Bob, but bob3 is a pre-existing reference, while friend is a reference received as part of the current sayHelloTo: message.



The fix in this case is simple -- we need to use the friend reference when sending both messages to Bob:

```
sayHelloTo: friend = (
  friend <-: hello.
  friend <-: goodbye.
)
```

The execution flow diagram now shows the messages to Bob appearing in the expected order.

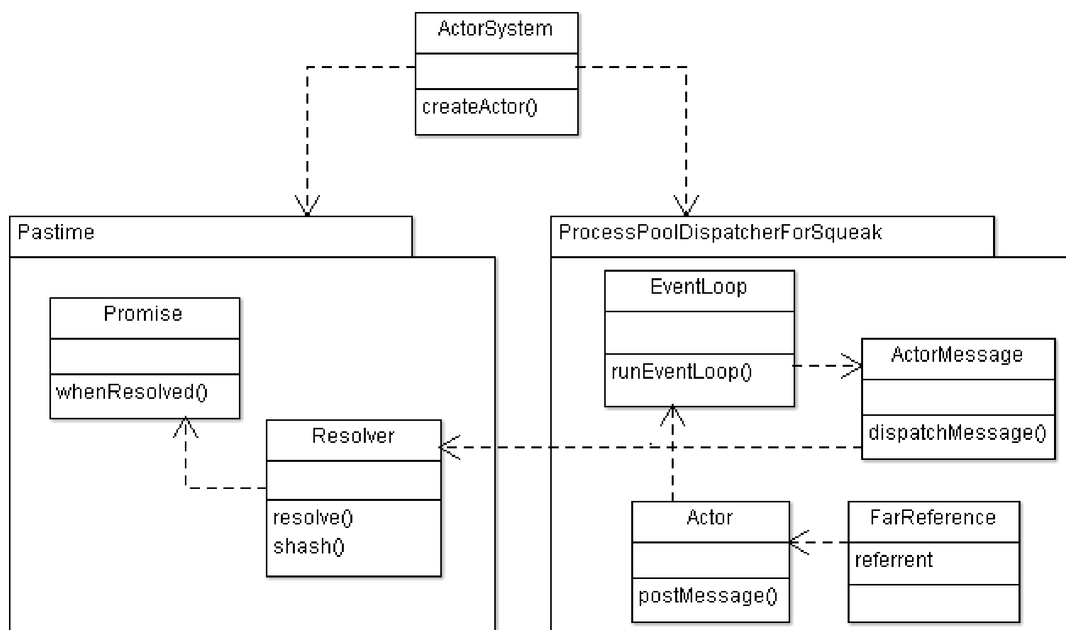


7 Design and Implementation

7.1 Design

7.1.1 Overview

The actor system implementation in Newspeak 4 consists of three primary modules -- Pastime, ProcessPoolDispatcherForSqueak and ActorSystem. The Pastime module implements E-style futures that function as an integral part of actors. The ProcessPoolDispatcherForSqueak module contains classes related to scheduling actor execution and actor message dispatch and message passing via far references. The ActorSystem module is the composition of the other two modules into a complete actor system. The following chart highlights the most important classes comprising the three modules and their interactions.



7.1.2 Actor System Portability

Because futures and actors are tightly integrated the `Pastime` and Dispatcher modules are mutually dependent. The motivation to separate the two modules is based on an important practical concern -- portability. The `Pastime` module contains pure Newspeak code and only relies on standard Newspeak libraries. The `ProcessPoolDispatcherForSqueak` module, however, imports implementation-specific classes, such as `Process`. As the name of the dispatcher module suggests, that module contains a dispatcher implementation for Newspeak-on-Squeak, which is the reference Newspeak implementation that runs on the Squeak Smalltalk VM and can interact with Smalltalk code running in the same VM. When porting Newspeak 4 and the actor system to another Newspeak implementation (e.g. Newspeak-on-Javascript or Newspeak-on-Dart), the `Pastime` module can be reused and only the implementation-specific dispatcher module needs to be re-implemented and provided for the new system. This follows the Newspeak best practice of isolating code that depends on the specific underlying implementation into separate modules.

7.1.3 Actor System Module Interactions

The createActor method of the `ActorSystem` module is the main API entry point to the Newspeak actor system. It accepts a Newspeak mixin object, clones the mixin object in a newly created actor's heap, applies the mixin to `Object`, and returns a far reference to the resulting class object. Instances of the `FarReference` class are a reification of far references in Newspeak. Each far reference is a tuple of two values -- a reference to the target object and a reference to the actor, in whose virtual heap the target resides. Eventual-sends to a far reference get routed to the `postMessage` method of the far reference's actor. Each actor is pinned to an EventLoop. An `EventLoop` is backed by a Squeak process (a green thread) and manages the execution of one or more actors. The actor's `postMessage` method wraps an eventual-send into an `ActorMessage` object and enqueues it to the actor's eventloop.

The `ActorMessage` class contains the implementation of actor message processing. Each `ActorMessage` object holds a reference to its target object, the message selector, arguments and a reference to the resolver, which is to receive the result of processing the message. `ActorMessage` instances get enqueued on an event loop's queue and are executed via their `dispatchMessage` method. The `dispatchMessage` method handles actor message processing, including passing on the return value or exception from a message execution to the messages' resolver. Each resolver object contains a reference to its corresponding promise object and its `resolve` and `smash` methods trigger the resolution of its promise.

7.1.4 GUI as an Actor

All code of a Newspeak 4 application runs inside the context of an actor. The first code that runs in a Newspeak 4 system is part of the GUI actor. To minimize the effort of integrating actors into Newspeak, the GUI actor is implemented by a custom bridge class -- `UIActor`, which implements the `postMessage` method by enqueueing the `ActorMessage` on the GUI event loop via Newspeak's existing `scheduleUIAction:` API. The `scheduleUIAction:` API is part of the Newspeak desktop system called Brazil, and is similar to the invokeLater API of Java Swing -- it takes a block closure and schedules it for execution on the GUI event loop.

7.1.5 Pastime and Past

The Newspeak platform includes a module called `Past`, which contains the implementations of `Delay`, `Future` and other classes needed to support lazy evaluation of simultaneous slots. The original design goal of `Past` was for actor futures to be included in the `Past` library and potentially even share the same future implementation between actors and simultaneous slots.

During the implementation of the actor framework for Newspeak, this author discovered that it is not feasible to share the same Future class for implementing simultaneous slots and actors. As a corollary of this, although possible, it does not seem desirable to keep actor futures and simultaneous slot futures in the same module. Hence the birth of the `Pastime` module, which is similar in goals, as in name to the `Past` module, but is dedicated to serving actors exclusively.

It is not feasible to use actor futures for simultaneous slots, because, as outlined in the previous sections, actor futures require tight coupling to the actor dispatcher module. This is because actor futures provide one of the two primitive means for scheduling computation on a Newspeak actor's mailbox. The first means is via eventual-sends, and the second means is the `whenResolved:` protocol on `Promise`. Furthermore, actor future promises, being eventual references, accept only eventual-sends, while futures used in the implementation of simultaneous slots must work in the purely sequential case, i.e. they must be able to accept immediate sends and function without the presence of actors or any other means of concurrency, for that matter.

7.2 Implementation

7.2.1 The Dispatcher

The dispatcher module is responsible for handling the message processing of all actors running inside a Newspeak VM. The dispatcher runs a pool of Squeak processes, each of which executes a single event loop. Each actor is pinned to an event loop process for its

entire lifetime. All actors belonging to the same event loop share that event loop's message queue. Assignment to event loops is done in a round-robin fashion at actor creation time. Since Squeak processes are green threads, the advantage to having a pool of processes is that actor execution can be interleaved, although there is no added parallelism. In order to increase interleaving even more, the dispatcher module yields execution to the next Squeak process at every eventual-send. The main advantage of this design is its simplicity and very low per-actor overhead. The primary drawback is that an actor taking a long time to complete one turn can potentially starve all other actors that share the same process. This issue can be eliminated by implementing actor stealing logic, where an idle event loop can steal idle actors from other processes and re-assign them to itself. However, since the Newspeak actor system is designed to encourage the use of many short-lived actors, this is not considered an issue of significant importance.

7.2.2 The Actor Class

One of the most interesting classes in the Newspeak actor system is the `Actor` class:

```
class Actor = (
"An actor class that is stateless and as lightweight as possible."
||) (
  postMessage: message <ActorMessage> = (
    messageQueue nextPut: message.
  )
) : ()
```

Note that the actor class is completely stateless -- it contains no slots. This is a conscious design decision based on the goal of minimizing the overhead of actor creation to the greatest extent possible. As mentioned in the previous section, each actor delegates its message processing to an `EventLoop` object. This means that an actor does not have its own queue, and therefore does not incur the overhead of queue object creation and storage. The main purpose that an actor object serves is to represent the unique identity of an actor -- different actor objects can be distinguished with the object identity comparison message (==).

The actor class does not need an explicit reference to its `EventLoop` object. The link to its associated `EventLoop` is implemented by nesting the `Actor` class inside the `EventLoop` class:

```
class EventLoop index: i <Integer> = (|
  private messageQueue <SharedQueue2> = SharedQueue2 new.
  … more slots …
|) (
  class Actor = ( … )
  … more instance members …
)
```

In Newspeak each instance of the `EventLoop` class carries its own instance of the `Actor` class, and the enclosing object reference is contained in the `Actor` class object, not in each `Actor` instance object, as is done for example for inner classes in Java. This means that the enclosing object reference is shared by all `Actor` objects and this results in further savings in the storage overhead per actor.

7.2.3 Argument Passing

There are two key operations involved in eventual-sends between actors. The first one is implementing argument passing semantics, which includes wrapping non-value object arguments in far references. This is done inside the `wrapArgs:from:to:` method of the dispatcher module:

```
wrapArgs: args <{}> from: srcActor <Actor> to: targetActor <Actor> = (
  ^args collect: [ :rawArg| | arg = pastime unwrapPromise: rawArg.|
    "1) Handle far references"
    (isFarReference: arg) ifTrue: [
      ".. unwrap far references to target actor ..."
    ]
    ifFalse: [
      "2) Handle promises"
      (pastime isPromise: arg) ifTrue: [
        "... create a new promise in the target actor chained to the
promise being passed ..."
      ]
      ifFalse: [
        "3) Handle value objects"
        (isValueObject: arg)
          ifTrue: [ "... pass by value ..." ]
          ifFalse: [ "... pass by far reference ..." ]
      ]
    ]
  ].
)
```

The `wrapArgs:from:to:` method takes a list of arguments, and a source and target actor. The source and target actor are assumed to be different, and the args are valid references in the source actor, which must be transformed into valid references in the target actor. There are three cases that must be handled -- far references, promises and value objects. Far references require special handling because a far reference from the source actor to the target actor must be unwrapped into a near reference in the target actor's heap. Promises must always be near references (local objects) and are handled by creating a new promise in the target actor and chaining it to the original promise from the source actor. The chaining is done by an inlined eventual-send of the `whenResolved:` message from the target actor to the promise in the source actor. Value objects are passed by reference since Newspeak actors run in a shared memory environment and value objects, which are deeply immutable can be safely passed directly between actor. All other objects are wrapped in a `FarReference` and the far reference is passed along.

The exclusive use of local promises is a simplification over the original model of Promises in E. E distinguishes between local and remote promise states. This allows for a more efficient implementation and facilitates the implementation of promise pipelining, and distributed actors.

Pipelined (batch) *delivery* of pipelined messages is an optimization that is transparent to the user and most critical in the distributed case because it reduces the number of round-trips across the network. Since distributed actors are not included in this initial implementation of Newspeak 4 actors, remote promises and pipelined delivery are also not supported. The implementation can be enhanced with those features in the future without impact on the semantics of actors and promises from the point of view of the actor system user.

7.2.4 Promise resolution

The second key operation of message passing in Newspeak is promise resolution. This is implemented in the `resolve:` method of the `Resolver` class:

```
resolve: resolution <Object> ^ <Boolean> = (
  | unresolved = promise state == 'unresolved'. |
  unresolved ifTrue: [
    (isPromise: resolution) ifTrue: [
      "Detect divergence."
      resolution == promise ifTrue: [ Error signal: 'divergence' ].
      resolution
        whenResolved: [ :finalResolution | resolve: finalResolution ]
        catch: [ :error | smash: error ] resolver: nil.
    ]
    ifFalse: [
      promise doResolve: resolution type: 'resolved'.
    ].
  ].
  ^unresolved.
)
```

First, the `resolve:` method ensures that the promise can be resolved only once. Repeated calls to the resolve method are ignored. Then the `resolve:` method handles promise chaining transparently to the user. If a promise is resolved with another promise, then resolution is delayed by chaining this promise to the provided promise. If the resolution is any other type of reference then the promise `doResolve:` method is called, which forwards all messages buffered by the promise. This ensures that `whenResolved:` handlers and pipelined message sends are only delivered when a promise is fully resolved to a near or far reference.

Unresolved `Promise` objects buffer two types of messages. Chained messages are buffered in response to `whenResolved:` and `whenResolved:catch:`, and have a known receiver (the block closure to be executed upon resolution) but take the yet-unknown resolution as an argument. Pipelined messages are buffered in response to messages eventually-sent to the yet-unknown resolution but their arguments are known. In both cases the buffered messages must carry a resolver object in addition to their receiver (chained messages) or selector and arguments (pipelined messages). The resolver is needed because both types of messages immediately return a promise themselves that can be used to await the result of processing these delayed messages. Chained and pipelined messages are represented by the `ChainMessage` and `PipelineMessage` private nested classes of `Promise`. Both classes support the same protocol that is defined in the abstract `PromiseMessage` class and includes the `dispatch` message.

7.2.5 Value objects

Newspeak 4 does not yet include first-class support for value objects with enforcement of deep immutability at the language level. Instead, objects that wish to be treated as values and passed by value between actors must return true in response to the `isValueObject` message:

```
isValueObject ^ <Boolean> = (
  ^true.
)
```

It is up to the application developer to ensure that objects that advertise themselves as values are truly deeply immutable, or otherwise safe for sharing between actors running in the same address space.

The Newspeak platform object, which contains the standard Newspeak libraries, such as collections, is treated as a value object. The platform object contains internal implementation classes and objects some of which are by necessity mutable. Access to such mutable state required for Newspeak implementation is properly guarded by using traditional shared-state concurrency primitives such as locks to ensure that the platform can be safely and efficiently shared among actors running on the same Newspeak VM.

7.2.6 Asynchronous Control Structures

Unlike the E language, Newspeak follows in the tradition of Smalltalk and Self in that it lacks built-in control structures. Branching and loops are implemented as message sends to boolean and collection objects. This means that control structures transparently integrate with actor message passing and can be used in an asynchronous fashion via eventual-sends.

In the following example, the delete message returns a boolean indicating success or failure and is awaited via message pipelining:

```
(file <-: delete)
  <-: ifTrue:  [ transcript <-: print: ‘success’ ]
      ifFalse: [ transcript <-: print: ‘failure’ ].
```

The same example in E must use the built-in when statement to await the result of the delete operation because the built-in if statement expects a value that is immediately available.

The current limitation of asynchronous control structure use in Newspeak is that the receiver of an asynchronous control structure message must eventually resolve to a near reference. This is always the case for boolean objects since `true` and `false` are value objects and passed by value between actors. More care must be taken with loop control structures, which can operate on collections that can potentially be mutable.

The reason for the restriction to near references is that control structure messages always immediately call the block closures they take as arguments. Block closures are not value objects in Newspeak and are therefore always passed by far reference. Sending the `do:` message to a mutable collection in another actor, for example, will fail when the `do:` implementation attempts to immediately invoke the closure argument, which will be received as a far reference that only accepts eventual-sends.

8 Conclusion

This report described how actors combined with E-style futures have a great potential to address the problems of asynchrony and parallelism, which are of growing importance in computer science today. In addition, we demonstrated how the actor-based concurrency framework in Newspeak 4 takes the first steps towards providing such capabilities in the Newspeak system.

One of the main arguments in this report is that an actor-based concurrency framework captures a very common pattern in asynchronous programming -- the event loop, and that E-style futures can be used to restore asynchronous control flow that spans multiple independently executing event loops. This unique combination of actors and non-blocking futures originates in Mark Miller's work on the E programming language and is what Miller calls "communicating event loops" [13].

A similar view of the future, in which actors play an important role in solving the concurrency challenges we face today is also echoed by other thought leaders in the software industry. Joe Duffy of Microsoft, the author of "Concurrent Programming on Windows" and a lead developer of Parallel LINQ says [26]:

> The major shift we face will be that mainstream languages will start to incorporate more concurrency-safety -- immutability and isolation -- and the platform libraries and architectures will better support this style of software decomposition. OOP developers are accustomed to partitioning their program into classes and objects; now they will need to become accustomed to partitioning their program into asynchronous Actors that can run concurrently. Within this sea of asynchrony will lay ordinary imperative code, frequently augmented with fine-grained task and data parallelism.

References

[1] The Newspeak Programming Language. http://newspeaklanguage.org/ (accessed April 22, 2012).

[2] Gilad Bracha. Newspeak Programming Language Draft Specification Version 0.07. http://bracha.org/newspeak-spec.pdf (accessed April 22, 2012).

[3] Katherine Yelick. Multicore: Fallout of a Hardware Revolution. http://newscenter.lbl.gov/wp-content/uploads/2008/07/yelick-berkeleyview-july081.pdf (accessed April 22, 2012).

[4] Manycore and Multicore Workshop (UNC-CH Computer Science) 2007. http://gamma.cs.unc.edu/SC2007/ (accessed April 22, 2012).

[5] Barcelona Multicore Workshop 2011. http://www.bscmsrc.eu/media/events/barcelona-multicore-workshop-2011 (accessed April 22, 2012).

[6] 4th Workshop on Dependable Many-Core Computing (DMCC 2012). http://hpcs2012.cisedu.info/2-conference/workshops/workshop-02-dmcc (accessed April 22, 2012).

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. Communications of the ACM. Volume 51 Issue 1, January 2008.

[8] Richard Brown. Hadoop at home: large-scale computing at a small college. SIGCSE ‘09.

[9] Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems. http://arxiv.org/abs/1008.1459v23 (accessed December 22, 2011).

[10] Akka. http://akka.io/ (accessed April 22, 2012).

[11] Gul Agha. Actors: A Model Of Concurrent Computation In Distributed Systems. MIT Press Cambridge, MA, USA ©1986.

[12] Humus. http://www.dalnefre.com/wp/humus/ (accessed April 22, 2012).

[13] Mark Miller. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[14] Dale Schumacher. Futures and Capabilities. http://www.dalnefre.com/wp/2012/03/futures-and-capabilities/ (accessed March 12, 2012).

[15] Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, 1998.

[16] Rajesh K. Karmani, Amin Shali, Gul Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. PPPJ ’09, August 27–28, 2009, Calgary, Alberta, Canada.

[17] Brian Goetz. Java theory and practice: Urban performance legends, revisited. http://www.ibm.com/developerworks/java/library/j-jtp09275/index.html (accessed April 22,

2012).

[18] Phaser (Java Platform SE 7). http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Phaser.html (accessed April 22, 2012).

[19] Gilad Bracha, Peter Ahe, Vassili Bykov, Yaron Kashai, William Maddox and Eliot Miranda. *Modules as Objects in Newspeak.* Proceedings of the 24th European Conference on Object Oriented Programming, Maribor, Slovenia, June 21-25 2010. Springer Verlag LNCS 2010.

[20] Ryan Macnak. Resolving pipelined futures. Newspeak Programming Language Google Group. https://groups.google.com/forum/#!msg/newspeaklanguage/3O4ltd94KIQ/xtU5MKuwzj0J (accessed April 22, 2012).

[21] Nikolay Botev. LazyTest.scala. https://github.com/bono8106/bikove/blob/648e87bdb5d68c103454a17e7751e7643721fb78/src/stuff/LazyTest.scala (accessed April 22, 2012).

[22] Nikolay Botev. ScalaFutureNonBlockingAPI.scala. https://github.com/bono8106/akkafutures/blob/019f313f23b3e4e106ce0f3706cba7bd96bc2838/src/main/scala/scalafutures/ScalaFutureNonBlockingAPI.scala (accessed April 22, 2012).

[23] Gilad Bracha, David Ungar. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

[24] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. OOPSLA '86 Proceedings.

[25] Roger Whitney. CS535: Doc 13, Double Dispatch. http://www.eli.sdsu.edu/courses/spring03/cs535/notes/dispatch/dispatch.html (accessed April 22, 2012).

[26] Joe Duffy. Joe Duffy's Weblog: An interview with InfoQ. http://www.bluebytesoftware.com/blog/2011/06/01/AnInterviewWithInfoQ.aspx (accessed April 22, 2012).

[27] Joe Armstrong. *Programming Erlang. Software for a Concurrent World.* The Pragmatic Bookshelf. 2007. Version 2009-2-9.

[28] Martin Odersky. Lex Spoon, Bill Venners. *Programming in Scala.* Artima, Inc. 2008. First Edition.

[29] Herb Sutter. A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3), March 2005. http://www.drdobbs.com/web-development/184405990 (accessed April 24, 2012).

[30] Rúnar Bjarnason. Beyond Mere Actors. http://docs.google.com/present/view?id=ddmk3f43_63zpg3jcgz (accessed May 5, 2012).

Appendix I - Scope and Limitations of the Current Implementation

Eventual-send Operator

Support for the eventual-send operator is implemented by translating <-: to a unary send of the ESEND message at parse time. For instance, the following code:

```
actor <-: message.
```

gets compiled into:

```
actor ESEND message.
```

The ESEND method is defined in Kernel`Object and returns an eventual-send proxy -- an object that accepts immediate-sens and forwards them as eventual-sends to a designated target.

This approach only requires a small modification to the Newspeak grammar and save a large amount of work in:

- precisely specifying the behavior of eventual-sends for all types of message sends (unary, binary, keyword and cascaded) in the grammar,
- defining and generating new AST nodes for eventual-sends, and
- generating the correct bytecodes from the AST.

The limitations of the above approach include:

- Dangling eventual-sends without a message will compile into innocent no-ops instead of generating a compile-time error, as in:

  ```
  actor <-:.
  ```

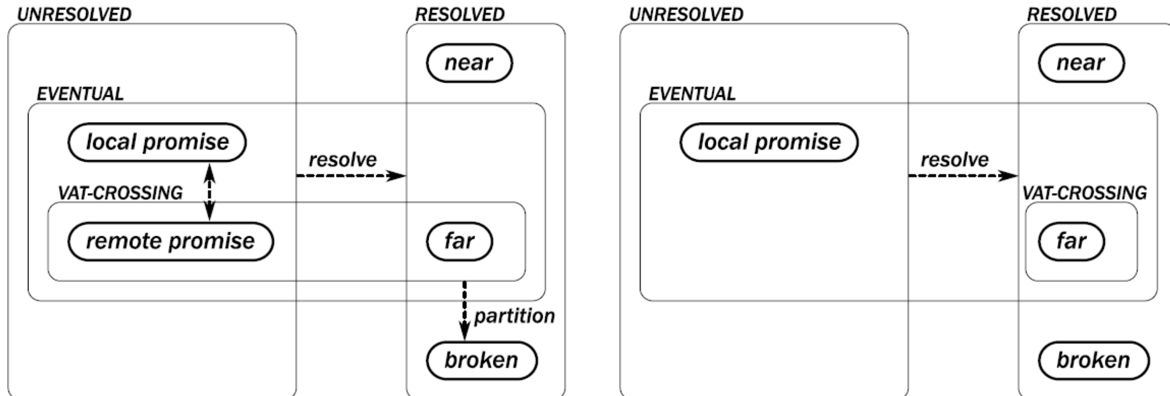
- Eventual-sends to literals (numbers, characters, strings, tuples, arrays) do not work, because literals in Newspeak still use the built-in Squeak classes, which do not extend from the Kernel`Object Newspeak class and therefore do not support the ESEND protocol.
- Carrying through of control structures will not work in some cases, because the compiler will fail to detect the eventual-send and will inline the control structure message. For example:

  ```
  | esendProxy = self <-: flipACoin ESEND. |
  esendProxy ifTrue: [ … ]. “error, the ifTrue: message was inlined.”
  ```

Promise Pipelining and Reference States

While promise pipelining is fully supported by the Promise implementation in Newspeak, multiple pipelined messages to the same vat are not delivered in a single round-trip, and this optimization cannot be implemented based on the current design. This is because, Promises do not currently distinguish between near and far unresolved states. The following figure compares

the reference states of E on the left vs Newspeak 4 on the right (source: Figure 17.1 in [13]):



There are two differences in the diagrams above -- Newspeak lacks the *remote promise* state and the transition from the vat-crossing states (*far* and *remote promise*) to the *broken* state (caused by partition). The first difference is critical to the actual pipelined *delivery* of pipelined eventual-sends -- the remote promise state holds the knowledge of the destination vat whose response is being awaited and to whom pipelined messages can be optimistically forwarded (in the expectation that the resolution resides in the destination vat where it can receive the pipelined messages without additional round-trips).

The second difference -- the lack of a transition from vat-crossing states to the broken state, is there because this transition is not needed. Actors that share the same address space cannot experience a partition, because the unit of failure is the Newspeak VM. Note that an unhandled exception does not cause an actor to be terminated. An actor lives as long as references to objects in its heap exist (which can originate either in its message queue or in far references from other actors). Partition can only occur between distributed actors that communicate over a network, and distributed actors are not supported by the current implementation.

From a functional perspective, the distinction between the local and remote promise states is completely transparent to the application code using actors and promises. Therefore, remote promise state can be introduced to the Newspeak actor framework implementation in the future without impact on existing code. Messages will be delivered more efficiently and in a fewer number of round-trips but the semantics of message delivery will remain the same from the point of view of the application. The lack of the partition transition represents missing functionality, which can also be added when support for distributed actors is implemented, along with its supporting APIs (something similar to _whenBroken in E would be needed) without any impact on existing code.

E-ORDER

E-ORDER is only really relevant in the distributed case when three or more actors communicate with each other over a network. In this scenario, for efficiency considerations a hypothetical implementation might decide that the act of enqueuing a message on a remote destination actor's queue will be asynchronous with respect to the eventual-send operator executed by the sending actor. This means that the eventual-send operator returns control to the sending actor *before* the message is guaranteed to have arrived in the recipient's queue. Without additional work by this hypothetical actor framework to enforce the constraints specified by E-ORDER, messages will be delivered in fail-stop FIFO (see Chapter 19 of [13]), which is weaker than E-

ORDER.

For actors that share an address space, it is usually more efficient to enqueue messages on the destination actor's queue synchronously, which leads to messages being delivered in CAUSAL order, which is stronger than E-ORDER. This is the case in the current actor implementation in Newspeak. E-ORDER is specified in this report because, even though Newspeak actors currently provide *accidental* CAUSAL order, the only guarantee as per the specification that developers can rely on is E-ORDER, and the implementation is free to evolve and provide weaker orders of message delivery (down to E-ORDER) in the future. In practice the implementation is likely to always deliver messages in CAUSAL order between actors within the same address space, and in E-ORDER between actors across address spaces. However, the scope of a Newspeak VM (which is presently not defined in the language specification, and no VM specification exists as of the time of this writing), while tied to a single address space in the current implementation, can evolve to span multiple address spaces in the future.

Value Objects

Value objects are not thoroughly defined in the Newspeak language specification (as of version 0.07) and not implemented at this time. The current actor implementation works around this issue by defining value objects as the set of the built-in literals nil, false, true, and all objects that advertise themselves as a Number, String, Character, or ValueObject. Furthermore, the Newspeak platform object (an instance of NewspeakRuntimeForSqueak) is tweaked to advertise itself as a ValueObject in order to facilitate sharing of the platform among actors. Without this modification, only the GUI actor would have direct access to the platform, which would make developing useful actos quite difficult.

For a truly useful actor system, an implementation of value objects is needed, which needs to include a list of built-in platform and implementation-specific classes and objects that are treated as value objects, even though they might hold mutable shared state, and are not necessarily serializable. Such objects protect their state using conventional locking primitives and are shared among actors of the same address space, and serialized “by key” between address spaces, and deserialized to the corresponding equivalent object that already exists in the target VM. This is a well-known issue related to serialization in general. Examples of such special objects include built-in Kernel class objects, such as String, Character, Object etc.

GUI as an Actor

The current implementation only provides the minimal necessary support for treating the GUI as an actor, which allows objects of the GUI actor to accept eventual-sends. A more thorough and radical implementation would include:

- Translating all existing GUI code (mostly in the IDE) that uses Squeak Processes for background tasks to use actors instead, and
- re-architecting the Newspeak GUI framework Brazil to send all user input and window system events as actor messages up the stack.

This could have implications on the design of the debugger infrastructure in Newspeak as well.

Debugging Actors

A minimal support is provided for debugging actors in the following form:

- halt in actor code will bring up the debugger. halt is a built-in message in Squeak, which signals the Halt exception. As a consequence, the Halt exception is treated specially in the actor framework and cannot be trapped by user code via Promise`whenResolved:catch:.
- Exceptions coming out of an eventual-send that are not handled by any whenResolved:catch: handler will bring up the debugger. The original activation stack that signaled the exception is lost, because passing along activation stacks in exceptions from one actor to another is not supported.

This is a far cry from full support for debugging actors. Some of the missing but desired functionality includes:

- the message queue of an actor is not available in the debugger;
- inspecting far references "sees through" the implementation and allows you to unsafely inspect the referent;
- passing along activation stacks in exceptions from one actor to another is not supported;
- all other actors pinned to the same dispatcher process as the actor whose code is being debugged will be suspended as well.

The core functionality required to implement some of the features is already available. For example, actor mirrors support inspecting an actor's queue of pending messages. Other functionality, such as passing along activation stacks in exceptions between actors could benefit from a complete Newspeak exceptions framework. Currently Newspeak relies directly on the Squeak exception library and Newspeak exceptions are not fully defined in the language specification. A definition of Newspeak exception should probably be done only after fully defining and implementing value objects, since exceptions that pass between actors must be first converted to, or treated implicitly as value objects.

Inspecting activation frames of an actor should ideally be done exclusively via eventual-sends by the debugger. This would allow the debugger to transparently support debugging of actors running in an address space different from that of the GUI actor executing the debugger.

The actor mirror API is currently only only a prototype, and the debugger is one of the major use cases for the actor mirrors and would be a driving factor in redefining and solidifying the API in the future.

Actor-based I/O APIs

Newspeak currently lacks an I/O library of its own for access to the filesystem, network etc. This is not a major issue in the absence of concurrency as Newspeak code can directly access Squeak classes and make use of the existing Squeak I/O libraries.

For a complete Newspeak 4 system where all code executes in the context of an actor, a purely asynchronous actor-based I/O library would be needed. Such a library can be implemented on top of the libuv cross-platform C library for non-blocking I/O, that is the foundation of the node.js

Javascript platform. libuv already comes with support for all major Operating Systems, including Windows, OS X and Linux, and is already designed with dynamic language bindings in mind (Javascript). The Newspeak Aliens API already provides full support for accessing libraries such as libuv from Newspeak on the Squeak platform, and thus libuv is likely the best candidate for use as the back-end of a non-blocking actor-based I/O API in Newspeak-on-Squeak.

Distributed Actors

The current implementation only supports actors running within the same Newspeak VM and address space. Actors distributed across Newspeak VMs and address spaces, and communicating over a network are not supported. An implementation of distributed actors depends on many of the aforementioned missing pieces:

- remote promises and the partition transition;
- E-ORDER guarantees (via forked references, as described in [13]);
- value objects;
- use of actor mirrors and eventual-sends for reflection by the debugger;
- actor-based I/O APIs.

Appending II - Bootstrapping the Actor Framework

All of the code of the actor framework implementation is available in the following two public bitbucket mercurial repositories:

- https://bitbucket.org/bono8106/nsactors
- https://bitbucket.org/bono8106/newspeak (**actors0** branch)

To bootstrap the actor framework on top of an existing Newspeak 3 image:

1. Use MemoryHole to pull in the code from the above two repositories into your image. From the newspeak repository in particular, you need the changes to the following classes:
    a. `Kernel`Object` (the `ESEND` method)
    b. `NewspeakRuntimeForSqueak` (the `actors` slot and `isValueObject` method)
    c. `Newspeak3Grammar`
    Note: run `` `Language resetNewspeak3` `` from a Squeak workspace, save the image and restart the IDE after loading the `Newspeak3Grammar` changes;
    d. `Newspeak3Compilation`Compiler`Rewriter`
2. Create the `ActorSystem` object from a workspace inspector:

```
ActorSystem usingPlatform: platform
  withDispatcher: ProcessPoolDispatcherForSqueak
  withPastime: Pastime withMirrors: ActorMirrors
```

3. Open an inspector on the newly-created `ActorSystem` object and run:

```
install
```

To run a few simple tests of the `ActorSystem`:

1. Create an `ActorSystemTests` object from a workspace inspector:

```
ActorSystemTests usingPlatform: platform
```

2. Open an inspector on the newly-created object and run:

```
testAll
```

The testAll method will create and run a few simple actors and produce output on the Squeak transcript.

There are currently no Minitest unit tests written for the actor framework.