What happened to ***Distributed Programming Languages?***

Heather Miller

PWLConf, September 28th, 2017, St. Louis, MO

Northeastern University

heather.miller.am/teaching/cs7680/index.html

CS 7680

Northeastern University
College of Computer and Information Science

Instructor: Heather Miller
heather at ccs dot neu dot edu
WVH242 (temp) & WVH302D
Fall 2016
Thursdays 6pm-9pm
Behrakis Center Room 204

Announcements
Schedule
Reading Research Papers
Weekly Tasks
Final Project

SPECIAL TOPICS IN COMPUTER SYSTEMS:

Programming Models for Distributed Computing

Distributed systems are everywhere today; games on smartphones are distributed as are popular web applications, and distributed batch and streaming computation on massive datasets is commonplace in industry today. To support this growing trend, much research in recent years has been devoted to the confluence of programming models and distributed systems. In this special topics course, we'll focus on that confluence, programming models for distributed computation, studying key publications describing systems and languages that have shaped how we build systems today, by improving reliability (fault tolerance), facilitating reasoning (correct-by-construction systems), and enabling better performance (reduced synchronization, in-memory computation). Some of these ideas have had a major impact in industry, and in these cases we'll highlight their relevance to building systems and applications today, while others have fallen short.

Topics we will cover include, promises, remote procedure calls, message-passing, conflict-free replicated datatypes, large-scale batch computation à la MapReduce/Hadoop and Spark, streaming computation, and where eventual consistency meets language design, amongst others.

Organization

The course is a research seminar that primarily focuses on

The Course

PhD Research Seminar at Northeastern University in the Fall of 2016.

PhD students, Master students, and even a few Bachelor students.

dist-prog-book.com/chapter/2/futures.html

Programming Models for
DISTRIBUTED COMPUTING

FUTURES AND PROMISES

Futures and Promises

BY KISALAYA PRASAD, AVANTI PATIL, AND HEATHER MILLER

Futures and promises are a popular abstraction for asynchronous programming, especially in the context of distributed systems. We'll cover the motivation for and history of these abstractions, and see how they have evolved over time. We'll do a deep dive into their differing semantics, and various models of execution. This isn't only history and evolution; we'll also dive into futures and promises most widely utilized today in languages like JavaScript, Scala, and C++. (53 min read)

—

Introduction

As human beings we have the ability to multitask *i.e.* we can walk, talk, and eat at the same time except when sneezing. Sneezing is a blocking activity because it forces you to stop what you're doing for a brief moment, and then you resume where you left off. One can think of the human sense of multitasking as multithreading in the context of computers.

Consider for a moment a simple computer processor; no parallelism, just the ability to complete one task or process at a time. In this scenario, sometimes the processor is blocked when some blocking operation is called. Such blocking calls can include I/O operations like reading/writing to disk, or sending or receiving packets over the network. And as programmers, we know that blocking calls like I/O can take a

The Book

Resulted in a book co-authored with the students in the course!

Muzammil Abdulrehman
Sam Caldwell
Nate Dempkowski
Tony Garnock-Jones
Aviral Goel
James Larisch
Fangfan Li
Abhilash Mysoresomashekar
Avanti Patil
Kisalaya Prasad
Jingjing Ren
Connor Zanin

Source on GitHub:
https://github.com/heathermiller/dist-prog-book

One thing I love about research papers
is that research is very animated.

But it doesn’t always look that way from the outside...


1
2
3


1
2
3

Name some distributed programming languages.

Name some distributed programming languages.

Erlang

Obliq

Lasp

Argus

Mirage

AliceML

Orca

Bloom

Linda

Emerald

Akka

Orleans

E

AmbientTalk

ML5

Name some distributed programming languages.

Erlang

Obliq

Lasp

Argus

Mirage

AliceML

Orca

Bloom

Linda

Emerald

Akka

Orleans

E

AmbientTalk

ML5

We'll touch on a handful today.

ARGUS
(1988)

SPECIAL SECTION

DISTRIBUTED PROGRAMMING IN ARGUS

Argus—a programming language and system developed to support the implementation and execution of distributed programs—provides mechanisms that help programmers cope with the special problems that arise in distributed programs, such as network partitions and crashes of remote nodes.

provides mechanisms that help programmers cope
- network partitions
- crashes of remote nodes

BARBARA LISKOV

Argus—a programming language and system—was developed to support the implementation and execution of distributed programs. Distribution gives rise to some problems that do not exist in a centralized system, or that exist in a less complex form. For example, a centralized system is either running or crashed, but a distributed system may be partly running and partly crashed. The goal of Argus is to provide mechanisms that make it easier for programmers to cope with these problems.

A program in Argus runs on one or more nodes. Each node is a computer with one or more processors and one or more levels of memory; we assume that the processors. Nodes can communicate with one another only by exchanging messages over the network. We make no assumptions about the network topology; for example, the network might be a local area net, or it might consist of a number of local area nets connected by a long haul net. In such a network, it is usually much faster for a node to access local information than information residing in some other node.

Distributed programs must cope with failures of the underlying hardware. Both the nodes and the network may fail. The only way nodes fail is by crashing; we assume it is impossible for a failed node to continue sending messages on the network. The network may lose messages or delay their delivery or deliver them out of order. It may also partition, so that some nodes are unable to communicate with other nodes for some period of time. In addition, the network may corrupt messages, but we assume the corruption is detectable; this assumption is satisfiable with arbitrarily high probability by including redundant information in messages.

Argus is intended to be used primarily for programs that maintain online data for long periods of time, e.g., file systems, mail systems, and inventory control systems. These programs have a number of requirements. Online information must remain consistent in spite of failures and also in spite of concurrent access. Programs must provide some level of service even when components fail; for example, a program may replicate information at several nodes so that individual failures can be masked. Programmers need to place information and processing at a particular node, both to do replication properly, and to improve performance, since information is cheaper to access if it is nearby. Finally, programs may need to be reconfigured dynamically, by adding and removing components

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, and monitored by the Office of Naval Research under Contract N00014-83-K-0125, and in part by the National Science Foundation under Grant DCR-8503662.

Observation:

> Distribution gives rise to some problems that do not exist in a centralized system or that exist in a less complex form. For example, a centralized system is either running or crashed, but a distributed system may be partly running and partly crashed.

Observation:

> Distribution gives rise to some problems that do not exist in a centralized system or that exist in a less complex form. For example, a centralized system is either running or crashed, but a distributed system may be partly running and partly crashed.

Nodes and network may fail.
Yet data must remain consistent!

Observation:

> Distribution gives rise to some problems that do not exist in a centralized system or that exist in a less complex form. For example, a centralized system is either running or crashed, but a distributed system may be partly running and partly crashed.

Nodes and network may fail.
Yet data must remain consistent!

MAJOR DESIGN DECISION!
STRONG CONSISTENCY IN THE LANGUAGE!

Observation:

> Distribution gives rise to some problems that do not exist in a centralized system or that exist in a less complex form. For example, a centralized system is either running or crashed, but a distributed system may be partly running and partly crashed.

Nodes and network may fail.
Yet data must remain consistent!

Problems with distribution

EXAMPLE: Banking

1. **Concurrent activities may interfere with one another.**
   For example, if a transfer runs concurrently with an audit, the audit might record a total that includes the withdrawal but not the deposit.

2. **Various failures are not taken into account.**
   For example, suppose the backend of the *to* account crashes immediately after the withdrawal from the *from* account has been made. In this case, we either need to put the money back into the *from* account or wait to complete the transfer until the *to* account's backend recovers.

Enter: Argus

Failures

2. *Various failures are not taken into account.* For example, suppose the back end of the *to* account crashes immediately after the withdrawal from the *from* account has been made. In this case we need to either put the money back into the *from* account or wait to complete the transfer until the *to* account's back end recovers.

ARGUS

Argus was designed to support programs like the banking system. To capture the object-oriented nature of such programs, it provides a special kind of object called a *guardian*, which implements a number of procedures that are run in response to remote requests. To solve the problems of concurrency and failures we have mentioned, Argus allows computations to run as *atomic transactions*, or *actions* for short. We will describe guardians and actions here; however more information can be found in [11, 12, 14]. We will illustrate their uses by showing how a portion of the banking system can be implemented in Argus.

Guardians

An Argus guardian is a special kind of abstract object whose purpose is to encapsulate a resource or resources. It permits its resource to be accessed by means of special procedures, called *handlers*, that can be called

Enter: Argus

Guardians

Special object. Runs procedures in response to remote requests.

Failures

2. *Various failures are not taken into account.* For example, suppose the back end of the *to* account crashes immediately after the withdrawal from the *from* account has been made. In this case we need to either put the money back into the *from* account or wait to complete the transfer until the *to* account's back end recovers.

ARGUS

Argus was designed to support programs like the banking system. To capture the object-oriented nature of such programs, it provides a special kind of object called a *guardian*, which implements a number of procedures that are run in response to remote requests. To solve the problems of concurrency and failures we have mentioned, Argus allows computations to run as *atomic transactions*, or *actions* for short. We will describe guardians and actions here; however more information can be found in [11, 12, 14]. We will illustrate their uses by showing how a portion of the banking system can be implemented in Argus.

Guardians

An Argus guardian is a special kind of abstract object whose purpose is to encapsulate a resource or resources. It permits its resource to be accessed by means of special procedures, called *handlers*, that can be called

Enter: Argus

Guardians

Special object. Runs procedures in response to remote requests.

Atomic Transactions

Computations organized as atomic transactions.

Failures

2. *Various failures are not taken into account.* For example, suppose the back end of the *to* account crashes immediately after the withdrawal from the *from* account has been made. In this case we need to either put the money back into the *from* account or wait to complete the transfer until the *to* account's back end recovers.

ARGUS

Argus was designed to support programs like the banking system. To capture the object-oriented nature of such programs, it provides a special kind of object called a *guardian*, which implements a number of procedures that are run in response to remote requests. To solve the problems of concurrency and failures we have mentioned, Argus allows computations to run as *atomic transactions*, or *actions* for short. We will describe guardians and actions here; however more information can be found in [11, 12, 14]. We will illustrate their uses by showing how a portion of the banking system can be implemented in Argus.

Guardians

An Argus guardian is a special kind of abstract object whose purpose is to encapsulate a resource or resources. It permits its resource to be accessed by means of special procedures, called *handlers*, that can be called

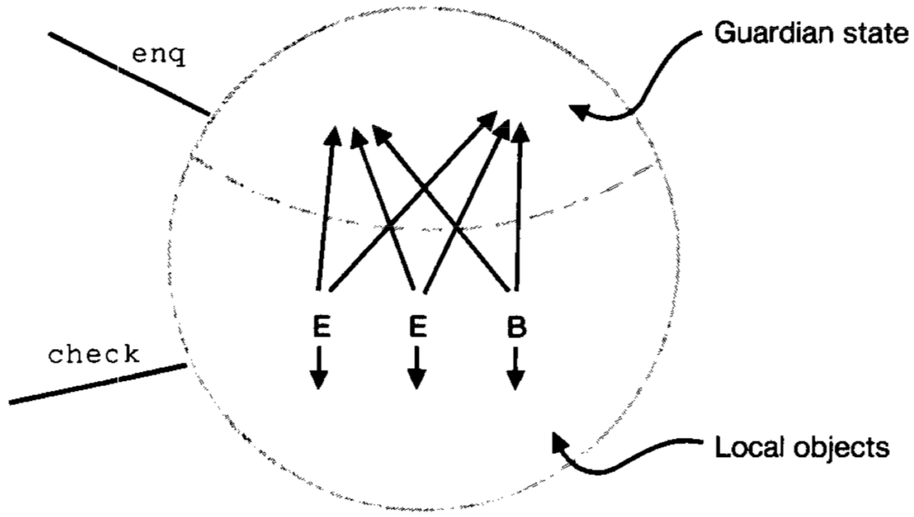
Guardians

> a special kind of abstract object
> **whose purpose is to encapsulate**
> a resource or resources.

Guardians

> a special kind of abstract object **whose purpose is to encapsulate** a resource or resources.

a guardian might control a printing device, and provide a handler called *enq* to allow files to be enqueued for printing and a handler called *check_queue* to check the state of the queue. A printer guardian is illustrated in Figure 3.

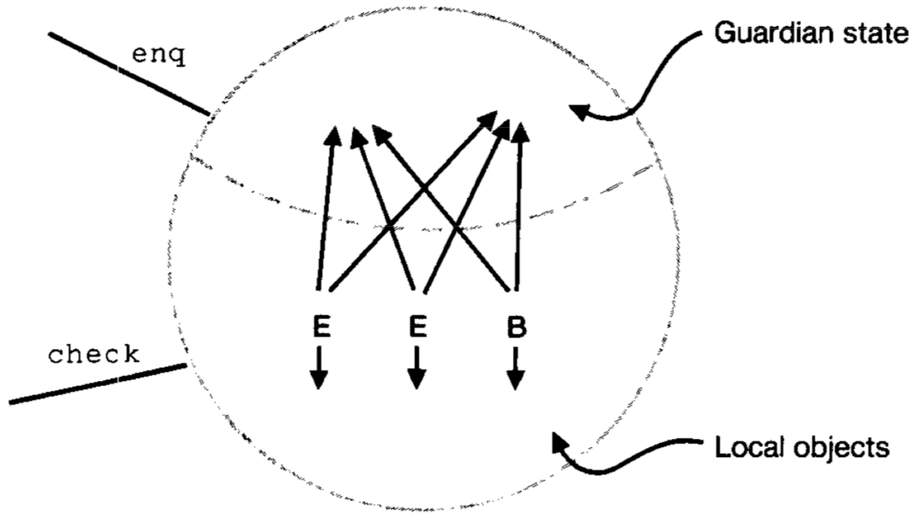


Guardians

> a special kind of abstract object **whose purpose is to encapsulate** a resource or resources.

1. Guardian encapsulates state. Contains dynamic collection of data objects.
2. Guardians are like an abstraction over a node. Can be moved.
3. A Guardian can create other Guardians dynamically.

a guardian might control a printing device, and provide a handler called *enq* to allow files to be enqueued for printing and a handler called *check_queue* to check the state of the queue. A printer guardian is illustrated in Figure 3.

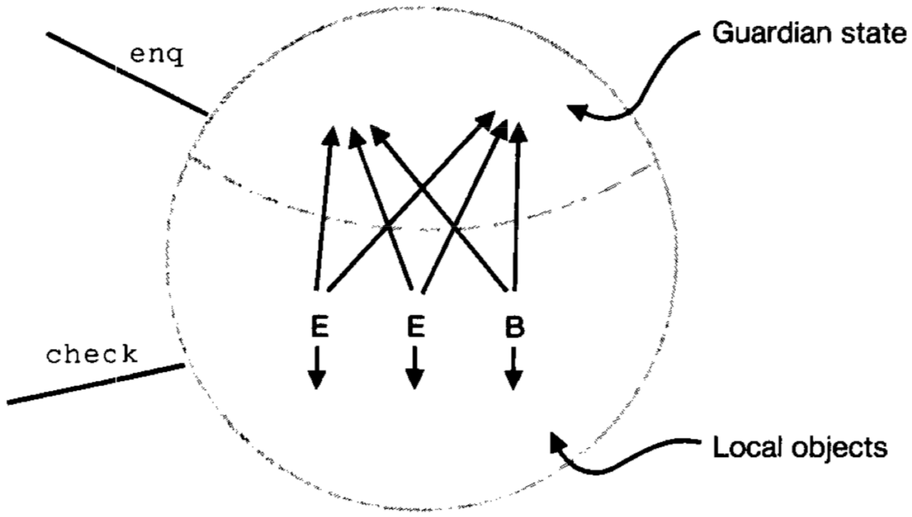


"Resilient" because guardian periodically writes stable objects to stable storage that it can read in again upon a failure.

Handlers

> A guardian permits its resources to be accessed by means of special procedures called *handlers*.

a guardian might control a printing device, and provide a handler called *enq* to allow files to be enqueued for printing and a handler called *check_queue* to check the state of the queue. A printer guardian is illustrated in Figure 3.

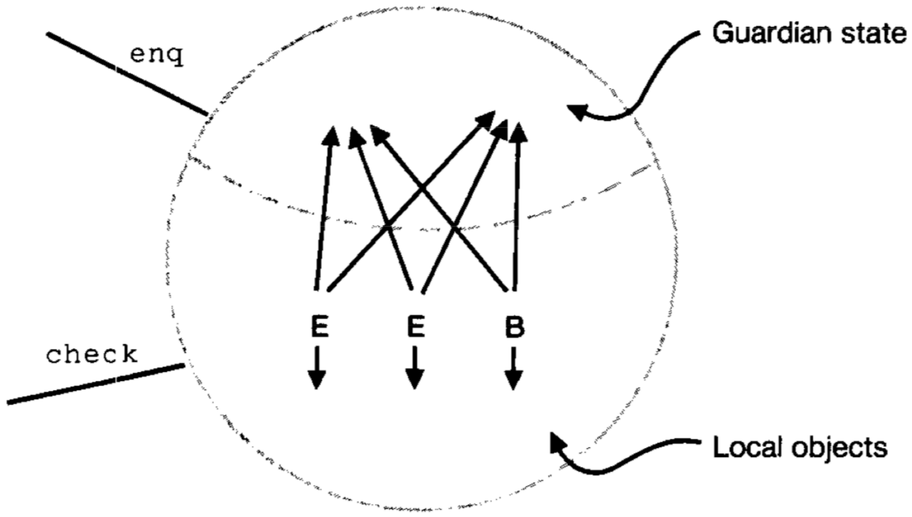


Handlers

> A guardian permits its resources to be accessed by means of special procedures called *handlers*.

1. **Guardian's data accessible only via calls to that Guardian's handler.**
2. **Handler calls location independent!**

a guardian might control a printing device, and provide a handler called *enq* to allow files to be enqueued for printing and a handler called *check_queue* to check the state of the queue. A printer guardian is illustrated in Figure 3.



Banking in Argus

This guardian implements the database for one branch. It maintains information about accounts in stable storage, and provides operations to open and close accounts, to withdraw and deposit money in accounts, and to provide the total of all accounts in the database.

FIGURE 6. The Branch Guardian

```
branch = guardian is create handles total, open, close, deposit, withdraw

  % type definitions
  htable = atomic_array[bucket]
  bucket = atomic_array[pair]
  pair = atomic_record[num: account_number, acct: acct_info]
  acct_info = atomic_record[bal: int]
  account_number = atomic_record[code: string, num: int]
  intcell = atomic_record[val: int]

  stable ht: htable          %  the table of accounts
  stable code: string        %  the code for the branch
  stable seed: intcell       %  the seed for generating new account numbers

  create = creator (c: string, size: int) returns (branch)
    code := c
    seed.val := 0
    ht := htable$new( )
    for i: int in int$from_to(1, size) do
      htable$addh(ht, bucket$new( ))
      end
    return (self)
    end create

  total = handler ( ) returns (int)
    sum: int := 0
    for b: bucket in htable$elements(ht) do
      for p: pair in bucket$elements(b) do
        sum := sum + p.acct.bal
        end
      end
    return (sum)
    end total

  open = handler ( ) returns (account_number)
    intcell$write_lock(seed)  % get a write lock on the seed
    a: account_number := account_number${code: code, num: seed.val}
    seed.val := seed.val + 1
    bucket$addh(ht[hash(a.num)], pair${num: a, acct: acct_info${bal: 0}})
    return (a)
    end open

  close = handler (a: account_number) signals (no_such_acct, positive_balance)
    b: bucket := ht[hash(a.num)]
    for i: int in bucket$indexes(b) do
      if b[i].num ~= a then continue end
      if b[i].acct.bal > 0 then signal positive_balance end
      b[i] := bucket$top(b)  % store topmost element in place of closed account
      bucket$remh(b)  % discard topmost element
      return
      end
    signal no_such_acct
    end close

  lookup = proc (a: account_number) returns (acct_info) signals (no_such_acct)
    for p: pair in bucket$elements(ht[hash(a.num)]) do
      if p.num = a then return (p.acct) end
      end
    signal no_such_acct
    end lookup

  deposit = handler (a: account_number, amt: int) signals (no_such_acct, negative_amount)
    if amt < 0 then signal negative_amount end
    ainfo: acct_info := lookup(a) resignal no_such_acct
    ainfo.bal := ainfo.bal + amt
    end deposit

  withdraw = handler (a: account_number, amt: int)
          signals (no_such_acct, negative_amount, insufficient_funds)
    if amt < 0 then signal negative_amount end
    ainfo: acct_info := lookup(a) resignal no_such_acct
    if ainfo.bal < amt then signal insufficient_funds end
    ainfo.bal := ainfo.bal - amt
    end withdraw

end branch
```

Banking in Argus

This guardian implements the database for one branch. It maintains information about accounts in stable storage, and provides operations to open and close accounts, to withdraw and deposit money in accounts, and to provide the total of all accounts in the database.

FIGURE 6. The Branch Guardian

branch guardian starts here

```
branch = guardian is create handles total, open, close, deposit, withdraw

    % type definitions
    htable = atomic_array[bucket]
    bucket = atomic_array[pair]
    pair = atomic_record[num: account_number, acct: acct_info]
    acct_info = atomic_record[bal: int]
    account_number = atomic_record[code: string, num: int]
    intcell = atomic_record[val: int]

    stable ht: htable          %  the table of accounts
    stable code: string        %  the code for the branch
    stable seed: intcell       %  the seed for generating new account numbers

    create = creator (c: string, size: int) returns (branch)
        code := c
        seed.val := 0
        ht := htable$new( )
        for i: int in int$from_to(1, size) do
            htable$addh(ht, bucket$new( ))
            end
        return (self)
        end create

    total = handler ( ) returns (int)
        sum: int := 0
        for b: bucket in htable$elements(ht) do
            for p: pair in bucket$elements(b) do
                sum := sum + p.acct.bal
                end
            end
        return (sum)
        end total

    open = handler ( ) returns (account_number)
        intcell$write_lock(seed)  % get a write lock on the seed
        a: account_number := account_number${code: code, num: seed.val}
        seed.val := seed.val + 1
        bucket$addh(ht[hash(a.num)], pair${num: a, acct: acct_info${bal: 0}})
        return (a)
        end open

    close = handler (a: account_number) signals (no_such_acct, positive_balance)
        b: bucket := ht[hash(a.num)]
        for i: int in bucket$indexes(b) do
            if b[i].num ~= a then continue end
            if b[i].acct.bal > 0 then signal positive_balance end
            b[i] := bucket$top(b)  % store topmost element in place of closed account
            bucket$remh(b)  % discard topmost element
            return
            end
        signal no_such_acct
        end close

    lookup = proc (a: account_number) returns (acct_info) signals (no_such_acct)
        for p: pair in bucket$elements(ht[hash(a.num)]) do
            if p.num = a then return (p.acct) end
            end
        signal no_such_acct
        end lookup

    deposit = handler (a: account_number, amt: int) signals (no_such_acct, negative_amount)
        if amt < 0 then signal negative_amount end
        ainfo: acct_info := lookup(a) resignal no_such_acct
        ainfo.bal := ainfo.bal + amt
        end deposit

    withdraw = handler (a: account_number, amt: int)
            signals (no_such_acct, negative_amount, insufficient_funds)
        if amt < 0 then signal negative_amount end
        ainfo: acct_info := lookup(a) resignal no_such_acct
        if ainfo.bal < amt then signal insufficient_funds end
        ainfo.bal := ainfo.bal - amt
        end withdraw

end branch
```

branch guardian ends here

Banking in Argus

This guardian implements the database for one branch. It maintains information about accounts in stable storage, and provides operations to open and close accounts, to withdraw and deposit money in accounts, and to provide the total of all accounts in the database.

FIGURE 6. The Branch Guardian

```
branch = guardian is create handles total, open, close, deposit, withdraw

  % type definitions
  htable = atomic_array[bucket]
  bucket = atomic_array[pair]
  pair = atomic_record[num: account_number, acct: acct_info]
  acct_info = atomic_record[bal: int]
  account_number = atomic_record[code: string, num: int]
  intcell = atomic_record[val: int]

  stable ht: htable          %  the table of accounts
  stable code: string        %  the code for the branch
  stable seed: intcell       %  the seed for generating new account numbers

  create = creator (c: string, size: int) returns (branch)
    code := c
    seed.val := 0
    ht := htable$new( )
    for i: int in int$from_to(1, size) do
      htable$addh(ht, bucket$new( ))
      end
    return (self)
    end create
```

These are all handlers

```
  total = handler ( ) returns (int)
    sum: int := 0
    for b: bucket in htable$elements(ht) do
      for p: pair in bucket$elements(b) do
        sum := sum + p.acct.bal
        end
      end
    return (sum)
    end total

  open = handler ( ) returns (account_number)
    intcell$write_lock(seed)  % get a write lock on the seed
    a: account_number := account_number${code: code, num: seed.val}
    seed.val := seed.val + 1
    bucket$addh(ht[hash(a.num)], pair${num: a, acct: acct_info${bal: 0}})
    return (a)
    end open

  close = handler (a: account_number) signals (no_such_acct, positive_balance)
    b: bucket := ht[hash(a.num)]
    for i: int in bucket$indexes(b) do
      if b[i].num ~= a then continue end
      if b[i].acct.bal > 0 then signal positive_balance end
      b[i] := bucket$top(b)  % store topmost element in place of closed account
      bucket$remh(b)  % discard topmost element
      return
      end
    signal no_such_acct
    end close

  lookup = proc (a: account_number) returns (acct_info) signals (no_such_acct)
    for p: pair in bucket$elements(ht[hash(a.num)]) do
      if p.num = a then return (p.acct) end
      end
    signal no_such_acct
    end lookup

  deposit = handler (a: account_number, amt: int) signals (no_such_acct, negative_amount)
    if amt < 0 then signal negative_amount end
    ainfo: acct_info := lookup(a) resignal no_such_acct
    ainfo.bal := ainfo.bal + amt
    end deposit

  withdraw = handler (a: account_number, amt: int)
          signals (no_such_acct, negative_amount, insufficient_funds)
    if amt < 0 then signal negative_amount end
    ainfo: acct_info := lookup(a) resignal no_such_acct
    if ainfo.bal < amt then signal insufficient_funds end
    ainfo.bal := ainfo.bal - amt
    end withdraw

end branch
```

Ok, so how is partial failure addressed by Guardians?

Ok, so how is partial failure addressed by Guardians?

Guardians allow programs to be decomposed into units of tightly coupled data and processing. However, they do not solve the synchronization and failure problems mentioned earlier. These problems are addressed by the second main mechanism in Argus, the atomic action.

Actions

> To solve the problems of concurrency and failures we have mentioned, Argus allows computations to run as **atomic transactions**, or actions for short.

Actions are:

1. **Serializable.**
   The effect of running a group of actions is the same as if they were run sequentially in some order.

2. **Total.**
   An action either completes entirely or it is guaranteed to have no visible effect.

Actions

> To solve the problems of concurrency and failures we have mentioned, Argus allows computations to run as **atomic transactions**, or actions for short.

indivisible!

Actions are:

1. **Serializable.**
   The effect of running a group of actions is the same as if they were run sequentially in some order.

2. **Total.**
   An action either completes entirely or it is guaranteed to have no visible effect.

Serializability solves the concurrency problem.

Totality solves the failure problem.

Actions

> To solve the problems of concurrency and failures we have mentioned, Argus allows computations to run as **atomic transactions**, or actions for short.

Actions are:

1. **Serializable.**
   The effect of running a group of actions is the same as if they were run sequentially in some order.

2. **Total.**
   An action either completes entirely or it is guaranteed to have no visible effect.

Serializability is achieved via synchronization.

Argus provides atomic (synchronized) data types like arrays, records, integers, etc.

All read/write locks.

Slow, but consistent!

Recovery is achieved via versioning.

Versions kept in volatile memory in order to be able to roll back in the event of a failure.

Communication between remote guardians is via RPC.

Bonus!

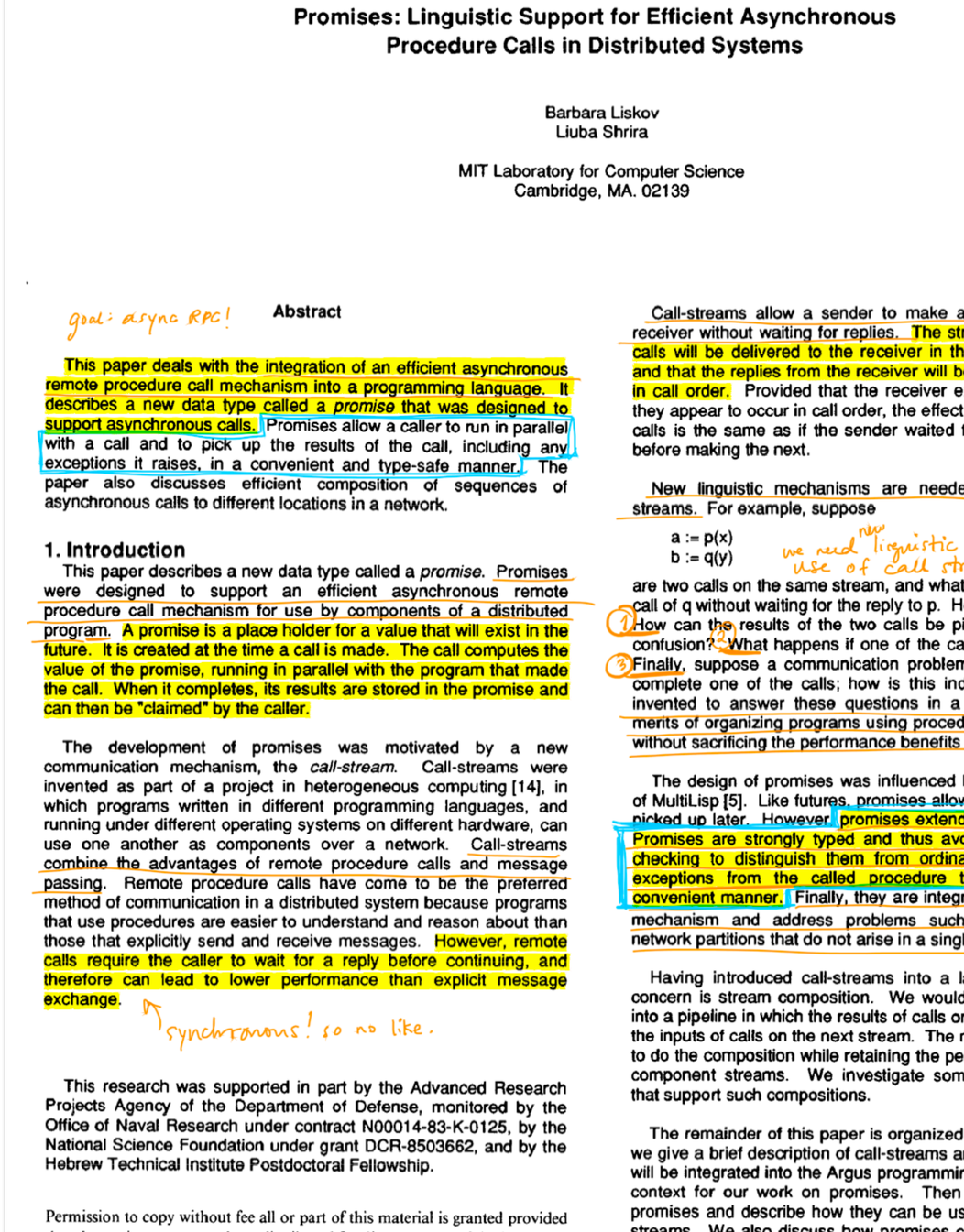
Promises came from Argus too!*

Promise pipelining came from Argus!

Promises provided a way to support asynchronous RPC calls.

+ strongly typed.

* Story of futures, promises, etc, a bit more involved. More info in the book!



Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems

Barbara Liskov
Liuba Shrira

MIT Laboratory for Computer Science
Cambridge, MA. 02139

goal: async RPC!

Abstract

This paper deals with the integration of an efficient asynchronous remote procedure call mechanism into a programming language. It describes a new data type called a *promise* that was designed to support asynchronous calls. Promises allow a caller to run in parallel with a call and to pick up the results of the call, including any exceptions it raises, in a convenient and type-safe manner. The paper also discusses efficient composition of sequences of asynchronous calls to different locations in a network.

1. Introduction

This paper describes a new data type called a *promise*. Promises were designed to support an efficient asynchronous remote procedure call mechanism for use by components of a distributed program. A promise is a place holder for a value that will exist in the future. It is created at the time a call is made. The call computes the value of the promise, running in parallel with the program that made the call. When it completes, its results are stored in the promise and can then be "claimed" by the caller.

The development of promises was motivated by a new communication mechanism, the *call-stream*. Call-streams were invented as part of a project in heterogeneous computing [14], in which programs written in different programming languages, and running under different operating systems on different hardware, can use one another as components over a network. Call-streams combine the advantages of remote procedure calls and message passing. Remote procedure calls have come to be the preferred method of communication in a distributed system because programs that use procedures are easier to understand and reason about than those that explicitly send and receive messages. However, remote calls require the caller to wait for a reply before continuing, and therefore can lead to lower performance than explicit message exchange.

synchronous! so no like.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, by the National Science Foundation under grant DCR-8503662, and by the Hebrew Technical Institute Postdoctoral Fellowship.

Call-streams allow a sender to make a
receiver without waiting for replies. The st
calls will be delivered to the receiver in th
and that the replies from the receiver will b
in call order. Provided that the receiver e
they appear to occur in call order, the effect
calls is the same as if the sender waited
before making the next.

New linguistic mechanisms are neede
streams. For example, suppose

```
a := p(x)
b := q(y)
```

we need new linguistic use of call st

are two calls on the same stream, and what
call of q without waiting for the reply to p. H
1 How can the results of the two calls be pi
confusion? 2 What happens if one of the ca
3 Finally, suppose a communication problem
complete one of the calls; how is this inc
invented to answer these questions in a
merits of organizing programs using proced
without sacrificing the performance benefits

The design of promises was influenced
of MultiLisp [5]. Like futures, promises allow
picked up later. However, promises extend
Promises are strongly typed and thus avo
checking to distinguish them from ordina
exceptions from the called procedure t
convenient manner. Finally, they are integr
mechanism and address problems such
network partitions that do not arise in a singl

Having introduced call-streams into a l
concern is stream composition. We would
into a pipeline in which the results of calls o
the inputs of calls on the next stream. The
to do the composition while retaining the pe
component streams. We investigate som
that support such compositions.

The remainder of this paper is organized
we give a brief description of call-streams a
will be integrated into the Argus programmi
context for our work on promises. Then
promises and describe how they can be us

How was Argus implemented?

Painstakingly.

Compiler closely coupled to kernel.

Fortunately no modifications to the Unix kernel though.

All networking, storage, etc, relies on a specific research implementation of Unix.

The current implementation of Argus is a prototype running on MicroVAX-IIs under Unix version Ultrix 1.2. The machines communicate over a 10 megabit/second ethernet. Each MicroVax has either 9 or 13 megabytes of primary memory and two RD53 disks, each with 70 megabytes of disk storage. One of the disks is used for our stable storage.

EMERALD
(1987)

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-13, NO. 1, JANUARY 1987

Distribution and Abstract Types in Emerald

ANDREW BLACK, NORMAN HUTCHINSON, ERIC JUL, HENRY LEVY, AND LARRY CARTER

Abstract—**Emerald is an object-based language for programming distributed subsystems and applications. Its novel features include 1) a single object model that is used both for programming in the small and in the large, 2) support for abstract types, and 3) an explicit notion of object location and mobility. This paper outlines the goals of Emerald, relates Emerald to previous work, and describes its type system and distribution support. We are currently constructing a prototype implementation of Emerald.**

Index Terms—**Abstract data types, distributed operating system, distributed programming, object-oriented programming, process migration, type checking.**

I. Introduction

WHILE distributed systems are now commonplace, the programming of distributed applications is still somewhat of a black art. We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on. Before the introduction of concurrent programming languages, concurrent programs were constructed in a similar fashion. Language support for concurrency greatly simplified concurrent programming; we believe that language support for distribution can have a similar effect on distributed programming. Experience with the remote procedure call facilities of Cedar/Mesa [2] and with the Eden Programming Language [1] has justified this belief. With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution.

Although distribution has many benefits [22], it also introduces challenges for the designer of a distributed language. First, the language must present a model of distributed computation; it must provide the conceptual framework that allows the programmer to define the objects that he manipulates in both the local and distributed environment. Second, it must provide for both intra- and internode communication in an efficient manner. The semantics of communication and computation should be consistent in the local and remote cases. Third, it must allow the programmer to exploit the inherent parallelism and availability of a distributed system. Fourth, since shutting down and recompiling an entire distributed system in order to modify some component is unacceptable, the language must permit system extensibility without recompilation; existing programs must continue to work in collaboration with new programs.

Our research focuses on simplifying the programming of distributed subsystems and applications by providing language support for distribution. We have designed an object-based language, called *Emerald*, and a distributed run-time system for Emerald that facilitate the construction of distributed programs for a local area network of independent nodes (workstations). The novel features of Emerald include: 1) a single object model that is used for both programming in the small and in the large, 2) support for abstract types, and 3) an explicit notion of object location and mobility. The goal of our research is to demonstrate the feasibility of using one simple semantic model for programming both sequential, single-node applications, and concurrent, potentially distributed applications. We currently have a prototype Emerald compiler and run-time system running on a local area network of VAX® workstations.

The next sections present a discussion of previous work in distributed programming languages and an overview of Emerald. Following sections describe the type system and the support for distribution.

II. Review of Previous Systems

To date, languages have supported distribution in several different ways. In the Xerox Cedar System [33], a remote procedure call facility allows programs to access remote servers through standard Cedar/Mesa procedure calls [2]. The advantage of this approach is that it requires no change to the semantics of the language. Automatically generated stub routines on the client and server machines are responsible for packing and unpacking parameters and transmitting and receiving messages. Programmers access a remote service in the same way that they would access a local service, except that they must explicitly locate and connect to the service before it can be used.

Manuscript received January 31, 1986; revised June 16, 1986. This work was supported in part by the National Science Foundation under Grants MCS-8004111 and DCR-8420945, by the University of Copenhagen, Denmark, under Grant J.nr. 574-2,2, and by a Digital Equipment Corporation External Research Grant.

A. Black, N. Hutchinson, E. Jul, and H. Levy are with the Department of Computer Science, University of Washington, Seattle, WA 98195.
L. Carter is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. This work was performed while he was a visitor at the University of Washington.

IEEE Log Number 8611363.

EMERALD
(1987)

Index Terms—Abstract data types, distributed operating system, distributed programming, object-oriented programming, process migration, type checking.

I. Introduction

WHILE distributed systems are now commonplace, the programming of distributed applications is still somewhat of a black art. We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on. Before the introduction of concurrent programming languages, concurrent programs were constructed in a similar fashion. Language support for concurrency greatly simplified concurrent programming; we believe that language support for distribution can have a similar effect on distributed programming. Experience with the remote procedure call facilities of Cedar/Mesa [2] and with the Eden Programming Language [1] has justified this belief. With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution.

Although distribution has many benefits [22], it also introduces challenges for the designer of a distributed language. First, the language must present a model of distributed computation; it must provide the conceptual framework that allows the programmer to define the objects that he manipulates in both the local and distributed

Manuscript received January 31, 1986; revised June 16, 1986. This work was supported in part by the National Science Foundation under Grants MCS-8004111 and DCR-8420945, by the University of Copenhagen, Denmark, under Grant J.nr. 574-2,2, and by a Digital Equipment Corporation External Research Grant

tem in order to modify some componer
the language must permit system exten
compilation; existing programs must c
collaboration with new programs.

Our research focuses on simplifying
of distributed subsystems and applicat
language support for distribution. We
object-based language, called *Emerald*
run-time system for Emerald that facil
tion of distributed programs for a loca
independent nodes (workstations). The
Emerald include: 1) a single object mo
both programming in the small and in th
for abstract types, and 3) an explicit n
cation and mobility. The goal of our r
onstrate the feasibility of using one simp
for programming both sequential, si
tions, and concurrent, potentially d
tions. We currently have a prototype
and run-time system running on a loc
VAX® workstations.

The next sections present a discussio
in distributed programming languages
Emerald. Following sections describe t
the support for distribution.

II. Review of Previous S

To date, languages have supported
eral different ways. In the Xerox Ce
remote procedure call facility allows
remote servers through standard Ced
calls [2]. The advantage of this approac
no change to the semantics of the la
cally generated stub routines on the cli
chines are responsible for packing and
eters and transmitting and receivin
grammers access a remote service in

EMERALD
(1987)

Index Terms—Abstract data types, distributed operating system, distributed programming, object-oriented programming, process migration, type checking.

I. Introduction

WHILE distributed systems are now commonplace, the programming of distributed applications is still somewhat of a black art. We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on. Before the introduction of concurrent programming languages, concurrent programs were constructed in a similar fashion. Language support for concurrency greatly simplified concurrent programming; we believe that language support for distribution can have a similar effect on distributed programming. Experience with the remote procedure call facilities of Cedar/Mesa [2] and with the Eden Programming Language [1] has justified this belief. With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution.

Although distribution has many benefits [22], it also introduces challenges for the designer of a distributed language. First, the language must present a model of distributed computation; it must provide the conceptual framework that allows the programmer to define the objects that he manipulates in both the local and distributed

Manuscript received January 31, 1986; revised June 16, 1986. This work was supported in part by the National Science Foundation under Grants MCS-8004111 and DCR-8420945, by the University of Copenhagen, Denmark, under Grant J.nr. 574-2,2, and by a Digital Equipment Corporation External Research Grant.

tem in order to modify some componer
the language must permit system exten
compilation; existing programs must c
collaboration with new programs.

Our research focuses on simplifying
of distributed subsystems and applicat
language support for distribution. We
object-based language, called *Emerald*
run-time system for Emerald that facil
tion of distributed programs for a loca
independent nodes (workstations). The
Emerald include: 1) a single object mo
both programming in the small and in th
for abstract types, and 3) an explicit n
cation and mobility. The goal of our r
onstrate the feasibility of using one simp
for programming both sequential, si
tions, and concurrent, potentially d
tions. We currently have a prototype
and run-time system running on a loc
VAX® workstations.

The next sections present a discussio
in distributed programming languages
Emerald. Following sections describe t
the support for distribution.

II. Review of Previous S

To date, languages have supported
eral different ways. In the Xerox Ced
remote procedure call facility allows
remote servers through standard Ced
calls [2]. The advantage of this approac
no change to the semantics of the la
cally generated stub routines on the cli
chines are responsible for packing and
eters and transmitting and receivin
grammers access a remote service in

~~EMERALD~~

~~(1987)~~

Index Terms—Abstract data types, distributed operating system, distributed programming, object-oriented programming, process migration, type checking.

I. Introduction

WHILE distributed systems are now commonplace, the programming of distributed applications is still somewhat of a black art. We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on. Before the introduction of concurrent programming languages, concurrent programs were constructed in a similar fashion. Language support for concurrency greatly simplified concurrent programming; we believe that language support for distribution can have a similar effect on distributed programming. Experience with the remote procedure call facilities of Cedar/Mesa [2] and with the Eden Programming Language [1] has justified this belief. With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution.

– Chris Meiklejohn, 2017

Although distribution has many benefits [22], it also introduces challenges for the designer of a distributed language. First, the language must present a model of distributed computation; it must provide the conceptual framework that allows the programmer to define the objects that he manipulates in both the local and distributed

Manuscript received January 31, 1986; revised June 16, 1986. This work was supported in part by the National Science Foundation under Grants MCS-8004111 and DCR-8420945, by the University of Copenhagen, Denmark, under Grant J.nr. 574-2,2, and by a Digital Equipment Corporation External Research Grant.

tem in order to modify some componer
the language must permit system exten
compilation; existing programs must c
collaboration with new programs.

Our research focuses on simplifying
of distributed subsystems and applicat
language support for distribution. We
object-based language, called *Emerald*
run-time system for Emerald that facil
tion of distributed programs for a loca
independent nodes (workstations). The
Emerald include: 1) a single object mo
both programming in the small and in th
for abstract types, and 3) an explicit n
cation and mobility. The goal of our r
onstrate the feasibility of using one simp
for programming both sequential, si
tions, and concurrent, potentially d
tions. We currently have a prototype
and run-time system running on a loc
VAX® workstations.

The next sections present a discussio
in distributed programming languages
Emerald. Following sections describe t
the support for distribution.

II. Review of Previous S

To date, languages have supported
eral different ways. In the Xerox Ced
remote procedure call facility allows
remote servers through standard Ced
calls [2]. The advantage of this approac
no change to the semantics of the la
cally generated stub routines on the cli
chines are responsible for packing and
eters and transmitting and receivin
grammers access a remote service in

EMERALD
(1987)

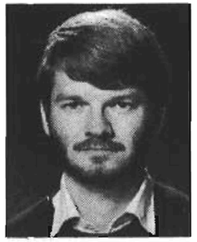
Andrew Black was born in London, England. He received the B.Sc. (Hons.) degree from the University of East Anglia in computing studies, and the D.Phil. degree from the Programming Research Group of the University of Oxford in programming languages and software engineering.

He has been on the faculty of the Department of Computer Science at the University of Washington, Seattle, since 1981. His current research interests are in the areas of distributed systems and programming language design, and in the way these areas interrelate.

Dr. Black is a member of the Association for Computing Machinery, the British Computer Society, the Union of Concerned Scientists, and Computer Professionals for Social Responsibility, and an affiliate of the IEEE Computer Society.

Norman Hutchinson received the B.Sc. degree in computer science from the University of Calgary, Calgary, Alta., Canada, in 1982, the M.S. degree in computer science from the University of Washington, Seattle, in 1985, and expects to receive the Ph.D. degree from the University of Washington in late 1986. His thesis topic is programming language design for distributed systems.

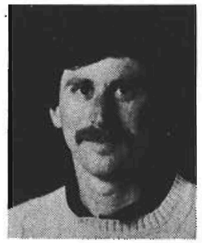
Effective January 1987, he will be an Assistant Professor of Computer Science at the University of Arizona in Tucson.



Eric Jul was born in Roskilde, Denmark. He received the Cand. Scient. (M.S.) degree from the University of Copenhagen, Copenhagen, Denmark, in 1980.

Currently, he is working toward the Ph.D. degree at the University of Washington, Seattle. His research interests are object-oriented systems, the design and implementation of programming languages for distributed systems, and distributed garbage collection. His thesis topic is object mobility in distributed systems.

Mr. Jul is a member of the Association for Computing Machinery, Computer Professionals for Social Responsibility, and the Danish Computing Society.



Henry Levy received the B.S. degree from Carnegie-Mellon University, Pittsburgh, PA, and the M.S. degree from University of Washington, Seattle.

He is Research Assistant Professor in the Department of Computer Science at the University of Washington and a Consulting Engineer on leave from Digital Equipment Corporation. His research interests include operating systems, distributed systems, and computer architecture. He is author of the book *Capability-Based Computer Systems* and coauthor of *Computer Programming and Architecture: The VAX-11*.

Larry Carter received the A.B. degree in mathematics from Dartmouth College, Hanover, NH, in 1969, and the Ph.D. degree in mathematics from the University of California at Berkeley in 1974.

Since 1975, he has been at IBM's Thomas J. Watson Research Center in Yorktown Heights, NY, except for sabbatical leaves to teach at the Penn State University and at Berkeley, and extended visits to the University of Washington in Seattle and to Hamsphire College in Amherst, MA. His research interests include the theory of computation and VLSI.

MOTIVATION:

Language Support for Distribution

PRECURSOR:

Eden language.

- Extension of Concurrent Euclid for distribution.

- Ended up with 2 object models
  - Eden's mobile objects very costly; too heavyweight (hundreds of milliseconds for basic messaging).
  - So, developers used objects from Concurrent Euclid and would communicate instead through shared memory instead.

THEREFORE:

New language called Emerald to address some of these issues.

MOTIVATION:

Language Support for Distribution

> We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on.

> With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution

MOTIVATION:

Language Support for Distribution

> We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on.

> With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution

Goals of Emerald

1. a single object model that is used for both programming in the small and in the large
2. support for abstract types
3. an explicit notion of object location and mobility

MOTIVATION:

Language Support for Distribution

> We believe that the complexity of distributed applications is heightened by the lack of programming language support for distribution. For example, most distributed applications are implemented by calling operating system communications primitives, such as send and receive. The programmer is responsible for locating the communications target, explicitly packaging parameters, and so on.

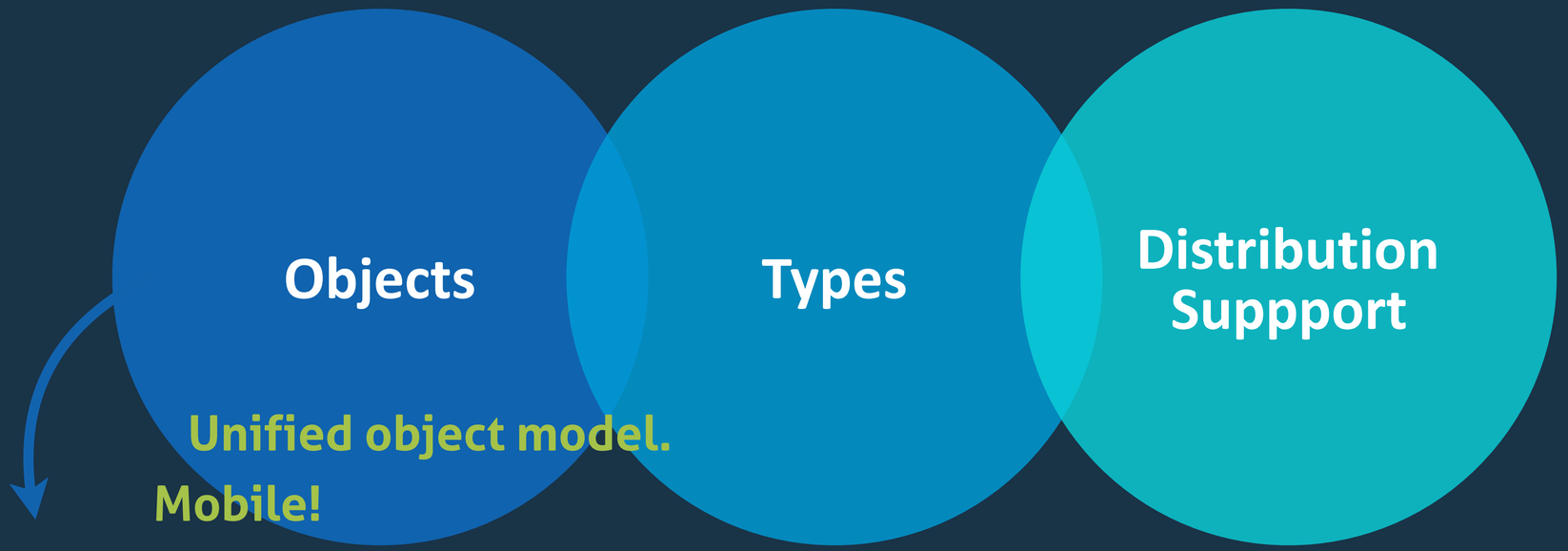
> With Emerald, we intend to go beyond simple syntactic support for message send and receive, and address some of the fundamental semantic problems of distribution

Goals of Emerald

& IS PERFORMANT!

1. a single object model that is used for both programming in the small and in the large
2. support for abstract types
3. an explicit notion of object location and mobility

Overview of Emerald



Consist of:
- Identity
- Representation
- Operations
- Process

+ Attributes:
- Location
- Immutable

+ Multiple object implementations selected by the compiler:
- Standard types (e.g., Integer)
- Local objects (local allocated data area)
- Movable/remotely referenced objects, implemented by a table for remote lookups.

Overview of Emerald

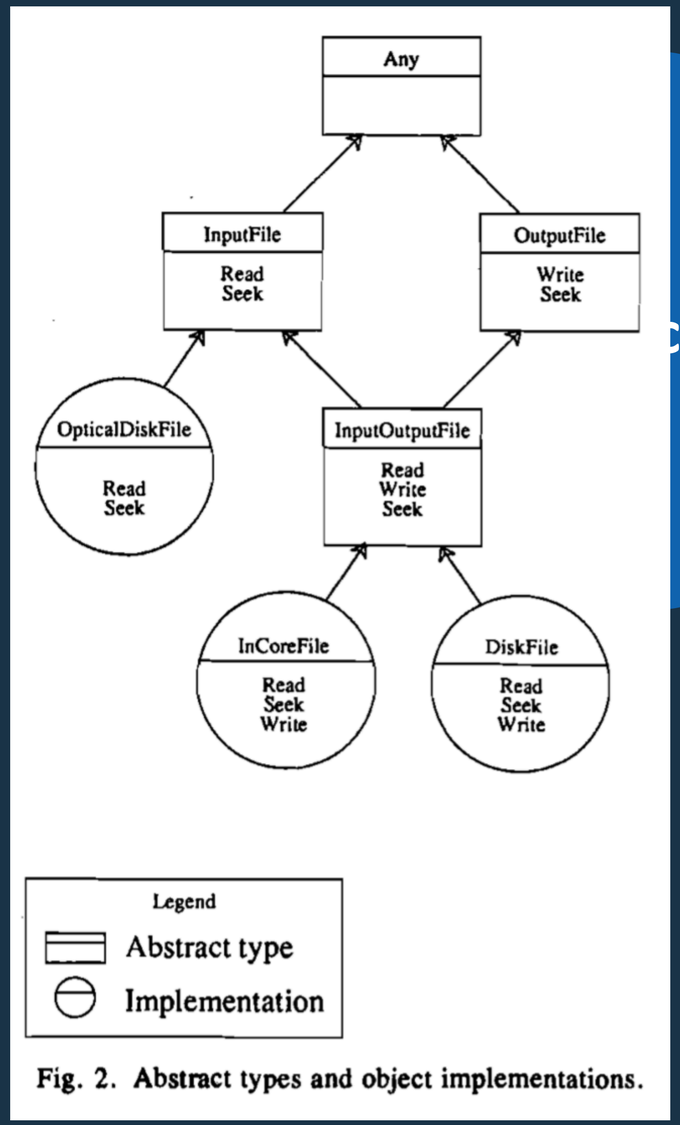


Fig. 2. Abstract types and object implementations.

cts

Types

Distribution
Suppport

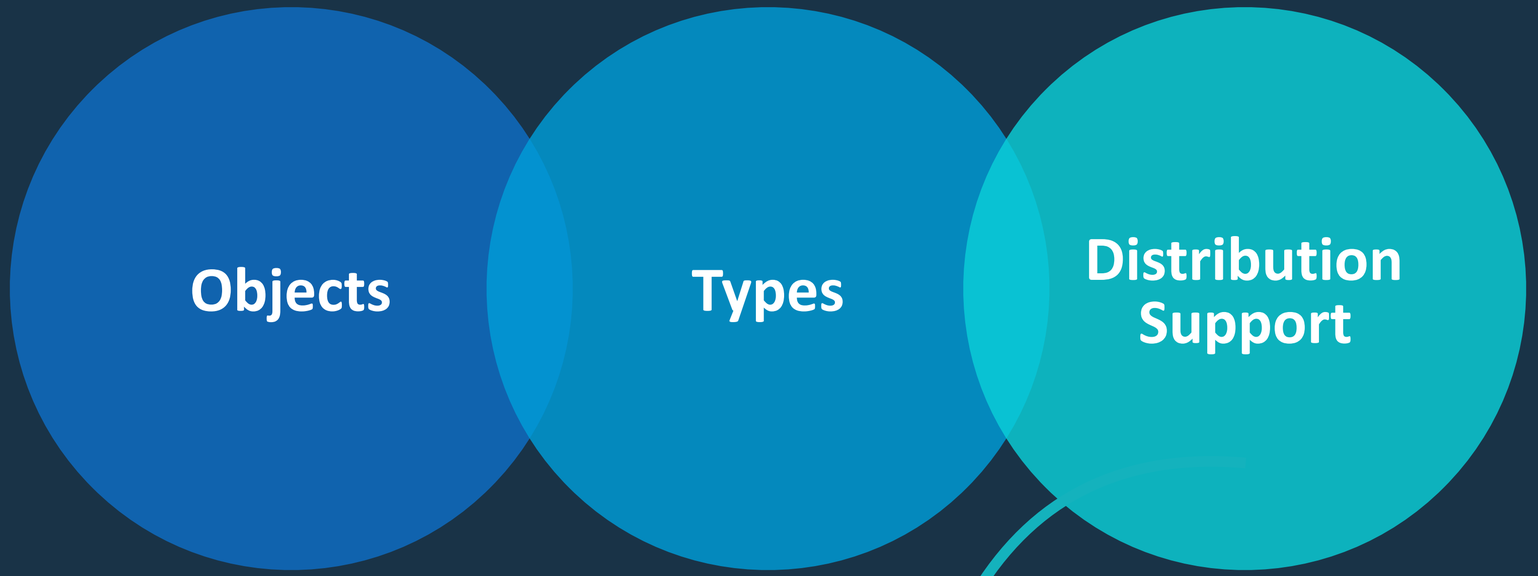
Abstract!

For better error messages
(as compared to SmallTalk)

Abstract types so Emerald could be used in open systems.

Where objects may be added to a running system.
Allow old code to invoke new objects w/ call to interface!

Overview of Emerald



Concurrency, nodes, and object location are integrated in to the Emerald language

- Objects located on specific nodes.
- Programmers can ignore/exploit object location.
- Applications are free to control the placement of objects.

argument motion, call-by-move, etc.

Rationale for object location visibility:

Performance & Availability

How was Emerald implemented?

Painstakingly.

Compiler very closely coupled to kernel.

Emerald defines an interface with kernel so that many operations can be performed either by the kernel, the compiler, or compiler generated code.

All networking, storage, etc, relies on a specific research implementation of Unix.

Only Emerald can run on the kernel.

The Emerald compiler and run-time system are currently able to support the creation and invocation of Emerald objects on a single processor or across the network. We are currently working on the implementation of object mobility and distributed garbage collection. The performance of local operations is summarized in Table II. The implementation of remote invocations is not yet at a stage where performance figures are meaningful.

ERLANG
(1990s)

DOI:10.1145/1810891.1810910

The same component isolation that made it effective for large distributed telecom systems makes it effective for multicore CPUs and networked applications.

BY JOE ARMSTRONG

Erlang

ERLANG IS A concurrent programming language designed for programming fault-tolerant distributed systems at Ericsson and has been (since 2000) freely available subject to an open-source license. More recently, we've seen renewed interest in Erlang, as the Erlang way of programming maps naturally to multicore computers. In it the notion of a process is fundamental, with processes created and managed by the Erlang runtime system, not by the underlying operating system. The individual processes, which are programmed in a simple dynamically typed functional programming language, do not share memory and exchange data through message passing, simplifying the programming of multicore computers.

Erlang[2] is used for programming fault-tolerant, distributed, real-time applications. What differentiates it from most other languages is that it's a concurrent programming language; concurrency belongs to the language, not to the operating system. Its programs are collections of parallel processes cooperating to solve a particular problem that can be created quickly and have only [illegible] overhead; programmers can create large numbers of Erlang processes yet ignore any preconceived ideas they might have about limiting the number of processes in their solutions.

All Erlang processes are isolated from one another and in principle are "thread safe." When Erlang applications are deployed on multicore computers, the individual Erlang processes are spread over the cores, and programmers do not have to worry about the details. The isolated processes share no data, and polymorphic messages can be sent between processes. In supporting strong isolation between processes and polymorphism, Erlang could be viewed as extremely object-oriented though without the usual mechanisms associated with traditional OO languages.

Erlang has no mutexes, and processes cannot share memory.[a] Even within a process, data is immutable. The sequential Erlang subset that executes within an individual process is a dynamically typed functional programming language with immutable state.[b] Moreover, instead of classes, methods, and inheritance, Erlang has modules that contain functions, as well as higher-order functions. It also includes processes, sophisticated error handling, code-replacement mechanisms, and a large set of libraries.

Here, I outline the key design criteria behind the language, showing how they are reflected in the language itself, as well as in programming language technology used since 1985.

Shared Nothing

The Erlang story began in mid-1985 when I was a new employee at the Ericsson Computer Science Lab in Stock-

a The shared memory is hidden from the programmer. Practically all application programmers never use primitives that manipulate shared memory; the primitives are intended for writing special system processes and not normally exposed to the programmer.

b This is not strictly

Erlang in 30 seconds

Functional Programming Language.

LIGHTWEIGHT PROCESSES:
The crux of Erlang.
Processes are typically created and live only long enough to handle a single request.
This allows countless processes to run at the same time.

MESSAGE-PASSING:
Processes communicate via messages.

IMMUTABLE:
Variables are assigned only once and can't be changed.

DYNAMIC:
VM designed for hot-swapping. Can upgrade applications w/out stopping them.

Erlang in 30 seconds

Functional Programming Language.

LIGHTWEIGHT PROCESSES:
The crux of Erlang.
Processes are typically created and live only long enough to handle a single request.
This allows countless processes to run at the same time.

MESSAGE-PASSING:
Processes communicate via messages.

IMMUTABLE:
Variables are assigned only once and can't be changed.

DYNAMIC:
VM designed for hot-swapping. Can upgrade applications w/out stopping them.

+OTP

Abstractions on top of Erlang that define applications.

Supervisors
Processes that watch over other processes and handle their failures. Child processes may in turn be supervisors themselves, forming a process hierarchy.

Servers
Processes that handle requests, can be synchronous or asynchronous.

State machines
Processes that live in one of a finite number of states and respond to events in possibly different ways depending on the current state.

summary from:
https://github.com/basho/riak_core/wiki/Erlang-OTP-in-a-Nutshell

How is Erlang implemented?

A Virtual Machine, compiled for Unix & Windows

Concurrency-focused VM.

Like a "process VM". No connection to OS processes or threads, but can manage millions of processes.

Garbage collector independently collects per process

Erlang processes don't share memory and have their own heaps, so they can be garbage collected independently.

Supports hot code reloading.

LINDA
(1994)







ELSEVIER

Parallel Computing 20 (1994) 633–655

PARALLEL COMPUTING

The Linda®† alternative to message-passing systems

Nicholas J. Carriero [a], David Gelernter [a], Timothy G. Mattson [b,‡], Andrew H. Sherman [c,*]

[a] *Department of Computer Science, Yale University, New Haven, CT 06520, USA*
[b] *Intel Supercomputer Systems Division, 14924 N.W. Greenbrier Parkway, Beaverton, OR 97006, USA*
[c] *Scientific Computing Associates, Inc. One Century Tower, 265 Church Street, New Haven, CT 06510-7010, USA*

(Received 17 May 1993; revised 25 November 1993)

Abstract

The use of distributed data structures in a logically-shared memory is a natural, readily-understood approach to parallel programming. The principal argument against such an approach for portable software has always been that efficient implementations could not scale to massively-parallel, distributed memory machines. Now, however, there is growing evidence that it *is* possible to develop efficient and portable implementations of virtual shared memory models on scalable architectures. In this paper we discuss one particular example: Linda. After presenting an introduction to the Linda model, we focus on the expressiveness of the model, on techniques required to build efficient implementations, and on observed performance both on workstation networks and distributed-memory parallel machines. Finally, we conclude by briefly discussing the range of applications developed with Linda and Linda's suitability for the sorts of heterogeneous, dynamically-changing computational environments that are of growing significance.

Key words: Message passing; LINDA; Virtual shared memory; Evaluation; Parallel programming paradigm

Linda makes shared memory possible!!

1. Introduction

Most of the papers in this special issue deal with message-passing libraries. Message passing is a coordination model that arises directly from the architecture

* *Corresponding author.*

† Linda is a registered trademark of Scientific Computing Associates, Inc.

‡ Dr. Mattson participated in this work prior to joining Intel, while he was affiliated with both Yale University and Scientific Computing Associates, Inc.

SSDI 0167-8191(93)E0102-2

What's wrong with message passing?

What's wrong with message passing?

Linda authors argue,

> *Message-passing is a coordination model that arises directly from the architecture of networks.*

What's wrong with message passing?

Linda authors argue,

> *Message-passing is a coordination model that arises directly from the architecture of networks.*

Yet,

> *The use of distributed data structures in a logically-shared memory is a natural, readily-understood approach to parallel programming.*

What's wrong with message passing?

Linda authors argue,

> *Message-passing is a coordination model that arises directly from the architecture of networks.*

Yet,

> *The use of distributed data structures in a logically-shared memory is a natural, readily-understood approach to parallel programming.*

Linda authors:

Guys, the "principal argument" against distributed shared memory is that "efficient implementations could not scale."

What's wrong with message passing?

Linda authors argue,

> *Message-passing is a coordination model that arises directly from the architecture of networks.*

Yet,

> *The use of distributed data structures in a logically-shared memory is a natural, readily-understood approach to parallel programming.*

Linda authors:

Guys, the "principal argument" against distributed shared memory is that "efficient implementations could not scale."

Claim:

- **message-passing was developed as a consequence of this complaint, in order to "cater to non-shared-memory architectures"**

What's wrong with message passing?

Linda authors argue,

> *Message-passing is a coordination model that arises directly from the architecture of networks.*

Yet,

> *The use of distributed data structures in a logically-shared memory is a natural, readily-understood approach to parallel programming.*

Linda authors:

Guys, the "principal argument" against distributed shared memory is that "efficient implementations could not scale."

Claim:

message-passing was developed as a consequence of this complaint, in order to "cater to non-shared-memory architectures"

Claim:

But scalable, efficient implementations of shared memory are possible!

Enter: Linda

Enter: Linda

BIG IDEA:

Tuplespaces!

virtual, associative (content-addressable), logically-shared memory

Enter: Linda

BIG IDEA:

Tuplespaces!

virtual, associative (content-addressable), logically-shared memory

tuplespaces contain...

Enter: Linda

BIG IDEA:

Tuplespaces!

virtual, associative (content-addressable), logically-shared memory

tuplespaces contain... Tuples!

Collection of ordered sequences of data called tuples.

Enter: Linda

BIG IDEA:

Tuplespaces!

virtual, associative (content-addressable), logically-shared memory

tuplespaces contain... Tuples!

Collection of ordered sequences of data called tuples.

For example:

```
('comment string', 1, 12, 4.99)
('logical data', .FALSE.)
('array data', [1 3 5 7])
```

Enter: Linda

BIG IDEA:

Tuplespaces!

virtual, associative (content-addressable), logically-shared memory

tuplespaces contain... Tuples!

Collection of ordered sequences of data called tuples.

For example:

```
('comment string', 1, 12, 4.99)
('logical data', .FALSE.)
('array data', [1 3 5 7])
```

Getting data in and out of tuplespaces:

CREATE / INSERT

serial:

```
out('table entry', i, j, f(i,j))
```

create & insert data serially via **out**

parallel:

```
eval('table entry', i, f(i))
```

create & insert data in parallel via **eval**

Enter: Linda

BIG IDEA:

Tuplespaces!

virtual, associative (content-addressable), logically-shared memory

tuplespaces contain... Tuples!

Collection of ordered sequences of data called tuples.

For example:

```
('comment string', 1, 12, 4.99)
('logical data', .FALSE.)
('array data', [1 3 5 7])
```

Getting data in and out of tuplespaces:

CREATE / INSERT

serial:

```
out('table entry', i, j, f(i,j))
```

create & insert data serially via **out**

parallel:

```
eval('table entry', i, f(i))
```

create & insert data in parallel via **eval**

Tuplespace is associative memory!

Tuples have no addresses. Selected for retrieval on the basis of any combination of their field values.

Enter: Linda

BIG IDEA:

Tuplespaces!

virtual, associative (content-addressable), logically-shared memory

tuplespaces contain... Tuples!

Collection of ordered sequences of data called tuples.

For example:

```
('comment string', 1, 12, 4.99)
('logical data', .FALSE.)
('array data', [1 3 5 7])
```

Getting data in and out of tuplespaces:

CREATE / INSERT

serial:

```
out('table entry', i, j, f(i,j))
```

create & insert data serially via **out**

parallel:

```
eval('table entry', i, f(i))
```

create & insert data in parallel via **eval**

Tuplespace is associative memory!

READ / REMOVE

read:

```
rd(A, ?w, ?x, ?y, ?z)
```

finds some tuple matching the pattern and yields it for destructuring and consumption by the invoking thread

remove:

```
in <pattern>
```

same as above + remove it from tuple space

Tuplespaces Good. QED.

Linda offers:
"intentionally loose coupling among processes"

whereas:
message-passing requires messages to be addressed to explicit receivers in point-to-point systems.

considered a win for Linda by the Linda designers.

Tuplespaces Good. QED.

Linda offers:
"intentionally loose coupling among processes"

whereas:
message-passing requires messages to be addressed to explicit receivers in point-to-point systems.

considered a win for Linda by the Linda designers.

Producers of tuples don't have to co-exist with consumers, since tuples remain in the tuple space until explicitly removed.

another win for Linda.

How was Linda implemented?

As a language extension!

C-Linda & Fortran-Linda

How was Linda implemented?

As a language extension!

C-Linda & Fortran-Linda

As a language extension, Linda comprises a small number of powerful operations that may be integrated into a conventional base programming language, yielding a dialect that supports parallel programming. Thus, for example, C and Fortran with the addition of the Linda operations become the parallel programming languages C-Linda and Fortran-Linda. SCIENTIFIC's implementations translate from the Linda parallel language (C-Linda or Fortran-Linda) into the corresponding base language (C or Fortran), automatically generate required auxiliary routines, and incorporate optimized kernel libraries to support the Linda operations at run-time. Portability comes from the consistency of the language processing between systems, while efficiency comes from the use of native C and Fortran compilers for the actual generation of object code, and from hardware-specific implementations of the kernels. Commercial versions of Linda now run well on a broad range of parallel computers, from shared-memory multiprocessors, to distributed-memory machines such as hypercubes, to networks of workstations. Since Linda has been discussed at length in the literature (e.g. [2,3,11,13]), we provide a relatively brief description here.

SCALA ACTORS
(2006)

Philipp Haller

Theoretical Computer Science 410 (2009) 202–220

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Scala Actors: Unifying thread-based and event-based programming☆

Philipp Haller*, Martin Odersky

EPFL, Switzerland

ARTICLE INFO

Keywords:
Concurrent programming
Actors
Threads
Events

ABSTRACT

There is an impedance mismatch between message-passing concurrency and virtual machines, such as the JVM. VMs usually map their threads to heavyweight OS processes. Without a lightweight process abstraction, users are often forced to write parts of concurrent applications in an event-driven style which obscures control flow, and increases the burden on the programmer.

In this paper we show how thread-based and event-based programming can be unified under a single actor abstraction. Using advanced abstraction mechanisms of the Scala programming language, we implement our approach on unmodified JVMs. Our programming model integrates well with the threading model of the underlying VM.

1. Introduction

Concurrency issues have lately received enormous interest because of two converging trends: first, multi-core processors make concurrency an essential ingredient of efficient program execution. Second, distributed computing and web services are inherently concurrent. Message-based concurrency is attractive because it might provide a way to address the two challenges at the same time. It can be seen as a higher-level model for threads with the potential to generalize to distributed computation. Many message passing systems used in practice are instantiations of the actor model [28,2]. A popular implementation of this form of concurrency is the Erlang programming language [4]. Erlang supports massively concurrent systems such as telephone exchanges by using a very lightweight implementation of concurrent processes [3,36].

On mainstream platforms such as the JVM [34], an equally attractive implementation was, as yet, missing. Their standard concurrency constructs, shared-memory threads with locks, suffer from high memory consumption and context-switching overhead. Therefore, the interleaving of independent computations is often modeled in an event-driven style on these platforms. However, programming in an explicitly event-driven style is complicated and error-prone, because it involves an inversion of control [41,13].

In previous work [24], we developed *event-based actors* which let one program event-driven systems without inversion of control. Event-based actors support the same operations as thread-based actors, except that the receive operation cannot return normally to the thread that invoked it. Instead the entire continuation of such an actor has to be a part of the receive operation. This makes it possible to model a suspended actor by a continuation closure, which is usually much cheaper than suspending a thread.

In this paper we present a unification of thread-based and event-based actors. An actor can suspend with a full thread stack (`receive`) or it can suspend with just a continuation closure (`react`). The first form of suspension corresponds to thread-based, the second form to event-based programming. The new system combines the benefits of both models.

full thread stack

closure

Actors modeled with closures

☆ A preliminary version of the paper appears in the proceedings of COORDINATION 2007, LNCS 4467, June 2007.

* Corresponding address: EPFL, Station 14, 1015 Lausanne, Switzerland. Tel.: +41 21 693 6483; fax: +41 21 693 6660.
E-mail address: philipp.haller@epfl.ch (P. Haller).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.09.019

Erlang + Scala? 🤔

Back in 2005 or 2006...

Can we bring these Erlang processes/actors to Scala somehow?

But Erlang has that nice “process VM”...

And Scala is on the JVM which has this super heavyweight threading model...

```
def awaitPing = react { case Ping => }
def sendPong = sender ! Pong
```

BIG IDEA:

Let’s do some of that process management that the Erlang VM does as a library! And map it down to JVM threads.

RESULT:

Lightweight threads à la Erlang, on top of the ForkJoin Pool.

This was a huge deal.

How were Scala Actors implemented?

as a regular Scala library!

Runs on an unmodified JVM.

Same is true for Akka.

Yieh!! They even look like they're keywords in the language. But they're just a library! Hah! Scala's so great 😍💓

CLOUD HASKELL (2011)

Towards Haskell in the Cloud

Jeff Epstein
University of Cambridge
jee36@cam.ac.uk

Andrew P. Black
Portland State University *
black@cs.pdx.edu

Simon Peyton-Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

Abstract

We present Cloud Haskell, a domain-specific language for developing programs for a distributed computing environment. Implemented as a shallow embedding in Haskell, it provides a message-passing communication model, inspired by Erlang, without introducing incompatibility with Haskell's established shared-memory concurrency. A key contribution is a method for serializing function closures for transmission across the network. Cloud Haskell has been implemented; we present example code and some preliminary performance measurements.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed Programming

General Terms Languages, Reliability, Performance

Keywords Haskell, Erlang, message-passing

1. Introduction

Cloud Haskell is a domain-specific language for cloud computing, implemented as a shallow embedding in Haskell. It presents the programmer with a computational model strongly based on the message-passing model of Erlang, but with additional advantages that stem from Haskell's purity, types, and monads.

The message-passing model, popularized by Erlang [1] for highly-reliable real-time applications and by MPI [6] for high-performance computing, stipulates that concurrent processes have no access to each other's data: any data that needs to be communicated from one process to another are sent explicitly in messages. We choose this model because it makes the costs of communication apparent, and because it makes a concurrent process a natural unit of failure: since processes do not share data, the data of one process cannot be contaminated by a fault in another.

We use the term "cloud" to mean a large number of processors with separate memories that are connected by a network and have independent failure modes. We don't believe that shared-memory concurrency is appropriate for programming the cloud. An effective programming model must be accompanied by a cost model. In a distributed memory system, the most significant cost, in both energy and time, is data movement. A programmer trying to reduce these costs needs a model in which they are explicit, not one that denies that data movement is even taking place — which is exactly the premise of a simulated shared memory.

One reason that this model of distributed computing has not previously been brought to Haskell is that it requires a way of running code on a remote system. Our work provides this, in the form of a novel method for serializing function closures. Without extending the compiler, our method hides the underlying serialization mechanism from the programmer, but makes serialization itself explicit.

In many ways, failure is *the* defining issue of distributed computation. In a network of hundreds of computers, some of them are likely to fail during the course of an extended computation; if our only recourse were to restart the computation from the beginning, the likelihood of it ever completing would become ever smaller as the system scales up. A programming system for the cloud must therefore be able to tolerate partial failure. Here again, Erlang has a solution that has stood the test of time; we don't innovate in this area, but adopt Erlang's solution (summarized in Section 2.5).

If Erlang has been so successful, you may wonder what Haskell brings to the table. The short answer is: purity, types, and monads. As a pure functional language, data is by default immutable, so the lack of shared, mutable data won't be missed. Importantly, immutability allows the implementation to decide whether to share or copy the data: the choice is semantically invisible, and so can depend on the locations of the processes. Moreover, pure functions are idempotent; this means that functions running on failing hardware can be restarted elsewhere without the need for distributed transactions or other mechanisms for "undoing" effects. Types in general, and monadic types in particular, help to guarantee properties of programs statically. For example, a function that has an externally-visible effect such as sending or receiving a message *cannot* have the same type as one that is pure. Monadic types make it convenient to program in an effectful style when that is appropriate, while ensuring that the programmer cannot accidentally mix up the pure and effectful code.

The contributions of this paper are as follows.

- A description of Cloud Haskell's interface functions (Sections 2 and 3). Following Erlang, our language provides a system for exchanging messages between lightweight concurrent processes, regardless of whether they are running on one computer or on many. We also provide functions for starting new remote processes, and for fault tolerance, which closely follow Erlang. However, unlike Erlang, Cloud Haskell also allows shared-memory concurrency *within* one of its processes.
- An additional message-passing interface that uses multiple *typed channels* in place of Erlang's single untyped channel (Section 4). Each channel is realized as a pair of ports; while the send port can be transmitted over the network, the receive port cannot.
- A method for serializing

* Research conducted while on sabbatical at Microsoft Research, Cambridge.

Haskell'11, September 22, 2011, Tokyo, Japan.

MOTIVATION:

Haskell extended to the cloud?

MOTIVATION:

Haskell extended to the cloud?

Simon Peyton Jones:

> I think there are really quite a lot of different paradigms for parallel programming in practice. And they differ mainly in their cost model. I'm not a believer in the one size fits all story about parallelism. I think we cannot escape the idea that we will need to write parallel programs using multiple different paradigms.

> I would like to build a language or language ecosystem where you can use lots of different kinds of parallelism in the same single application even. Where you have maybe bits of task parallelism, bits of data parallelism, and bits of message-passing all in the same application.

MOTIVATION:

Haskell extended to the cloud?

Simon Peyton Jones:

> I think there are really quite a lot of different paradigms for parallel programming in practice. And they differ mainly in their cost model. I'm not a believer in the one size fits all story about parallelism. I think we cannot escape the idea that we will need to write parallel programs using multiple different paradigms.

> I would like to build a language or language ecosystem where you can use lots of different kinds of parallelism in the same single application even. Where you have maybe bits of task parallelism, bits of data parallelism, and bits of message-passing all in the same application.

Looks to Erlang

Message-passing + Erlang programming model successful...
So let's bring it to Haskell!

MOTIVATION:

Haskell extended to the cloud?

But Haskell can do better!

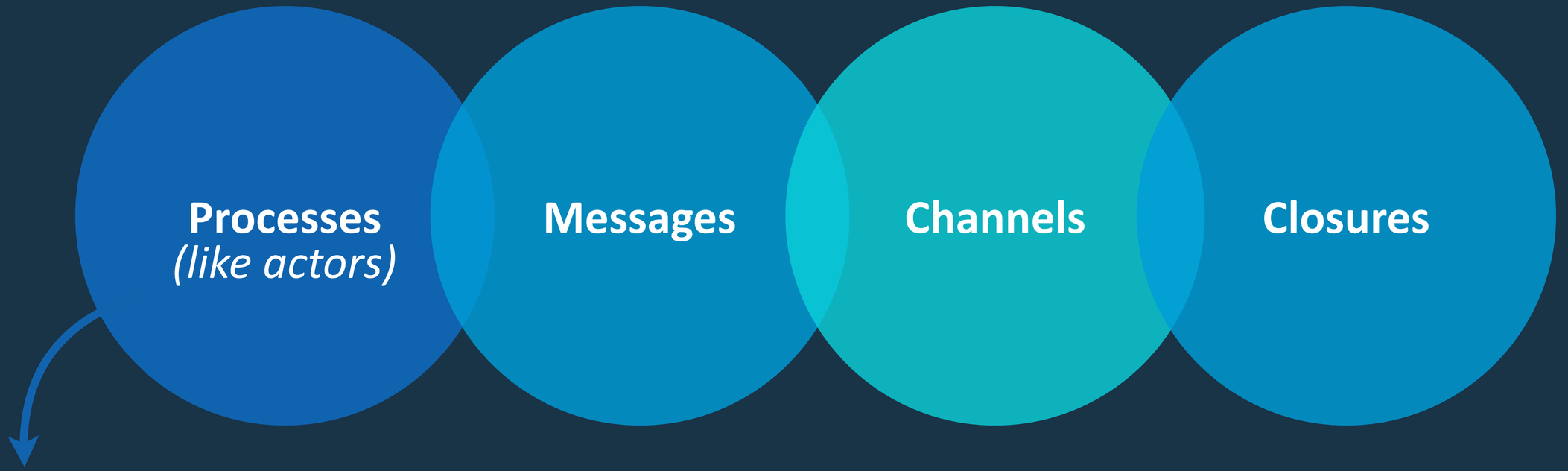
If Erlang has been so successful, you may wonder what Haskell brings to the table. The short answer is: purity, types, and monads. As a pure functional language, data is by default immutable, so the lack of shared, mutable data won't be missed. Importantly, immutability allows the implementation to decide whether to share or copy the data: the choice is semantically invisible, and so can depend on the locations of the processes. Moreover, pure functions are idempotent; this means that functions running on failing hardware can be restarted elsewhere without the need for distributed transactions or other mechanisms for "undoing" effects. Types in general, and monadic types in particular, help to guarantee properties of programs statically. For example, a function that has an externally-visible effect such as sending or receiving a message *cannot* have the same type as one that is pure. Monadic types make it convenient to program in an effectful style when that is appropriate, while ensuring that the programmer cannot accidentally mix up the pure and effectful code.

Immutability **Types**

fns idempotent
(good for fault tol.)

separate pure & effectful code with monads

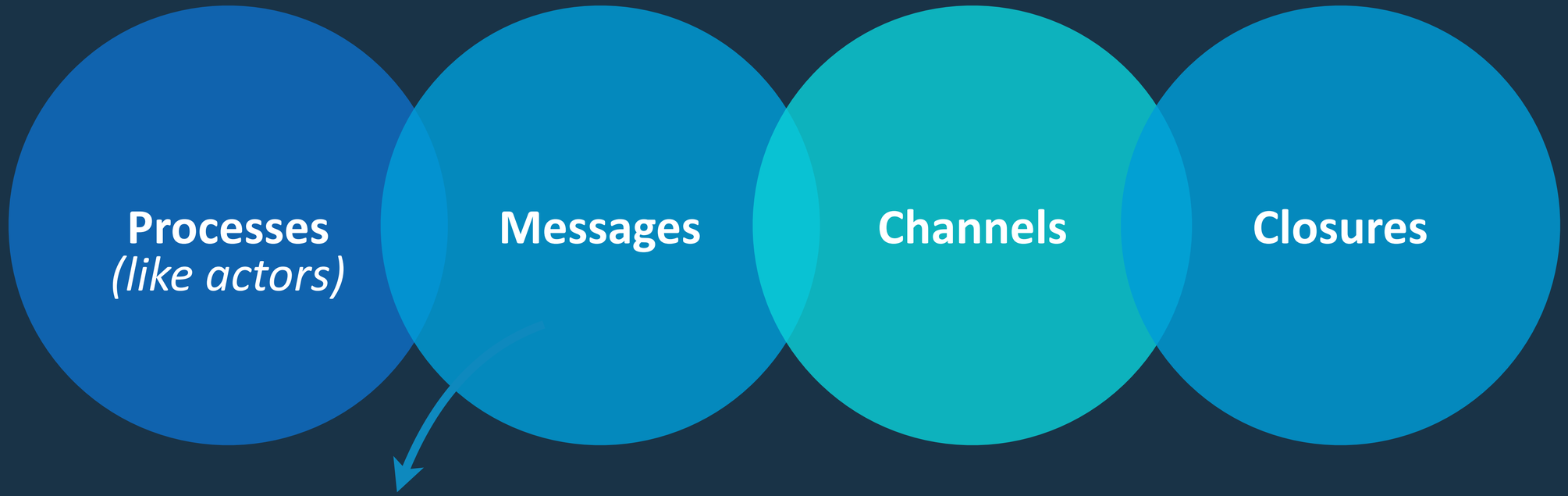
Cloud Haskell Overview



Process can send/receive messages of any type to another process via the following functions:

```
send :: Serializable a => ProcessId -> a -> ProcessM ()
expect :: Serializable a => a -> ProcessM ()
```

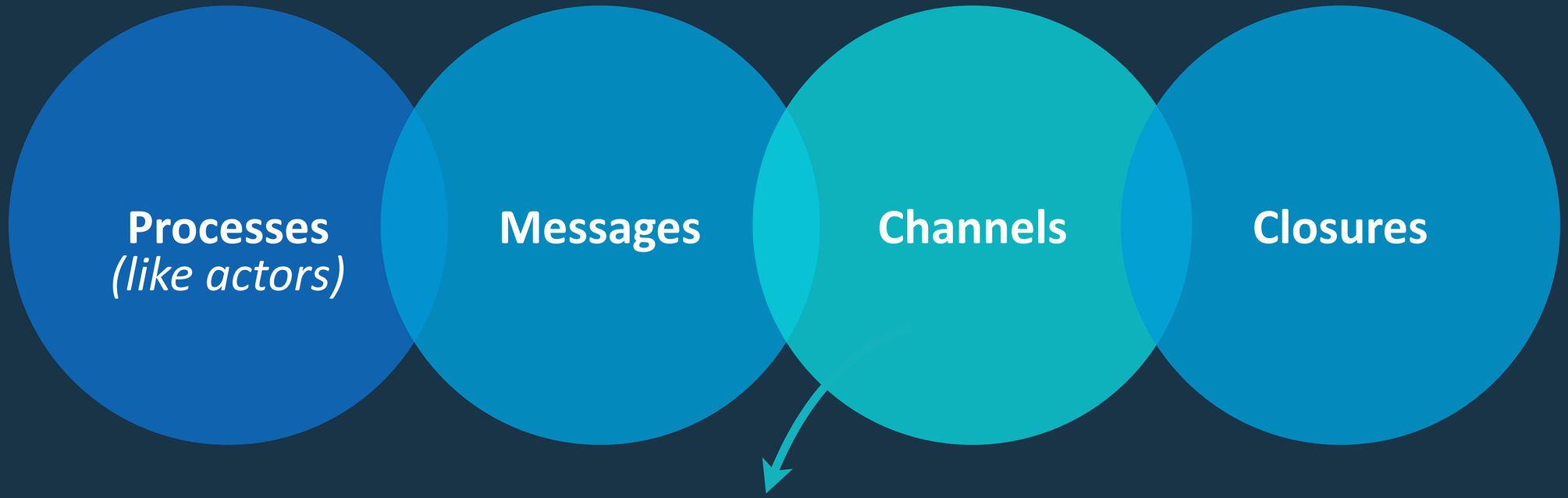
Cloud Haskell Overview



Messages a piece of information sent from one process to another. Must be instance of Serializable typeclass:

```
class (Binary a, Typeable a) => Serializable a
```

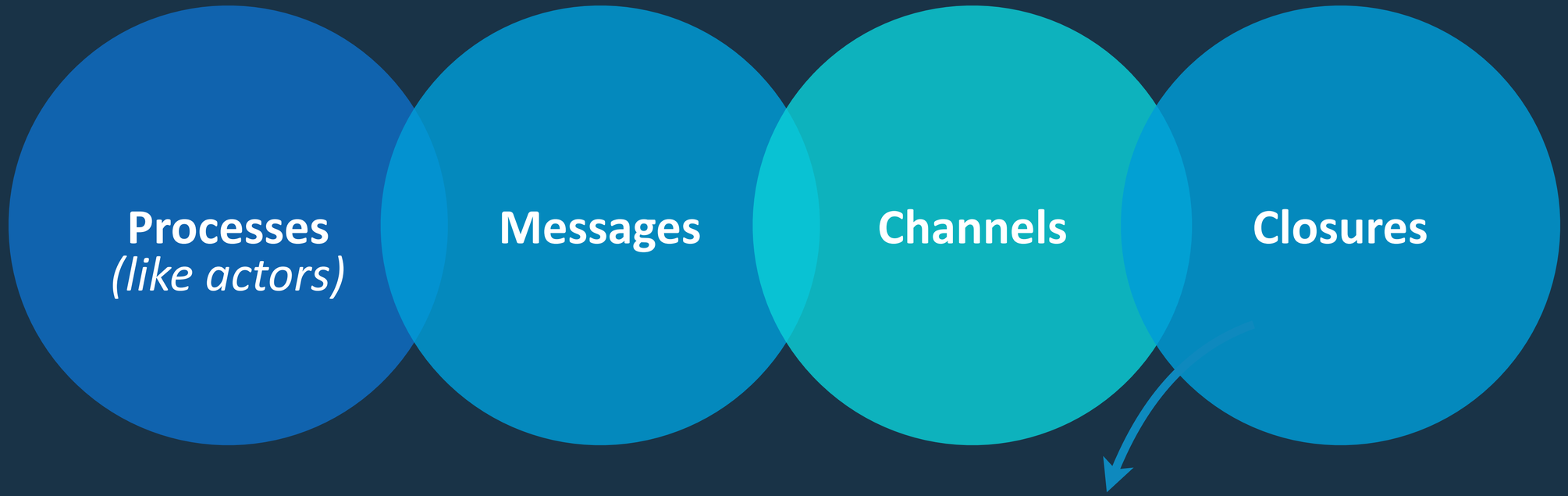
Cloud Haskell Overview



An explicit data structure used to send message of exactly one type to exactly one node.

```
newChan :: Serializable a => ProcessM (SendPort a, ReceivePort a)
```

Cloud Haskell Overview



Closures in Cloud Haskell are restricted. Only free variables that are statically-bound or variables defined at the top-level are valid environments.

How was Cloud Haskell implemented?

As a domain-specific language!

Library + a little bit of Template Haskell

Ping Pong in Cloud Haskell vs Erlang:

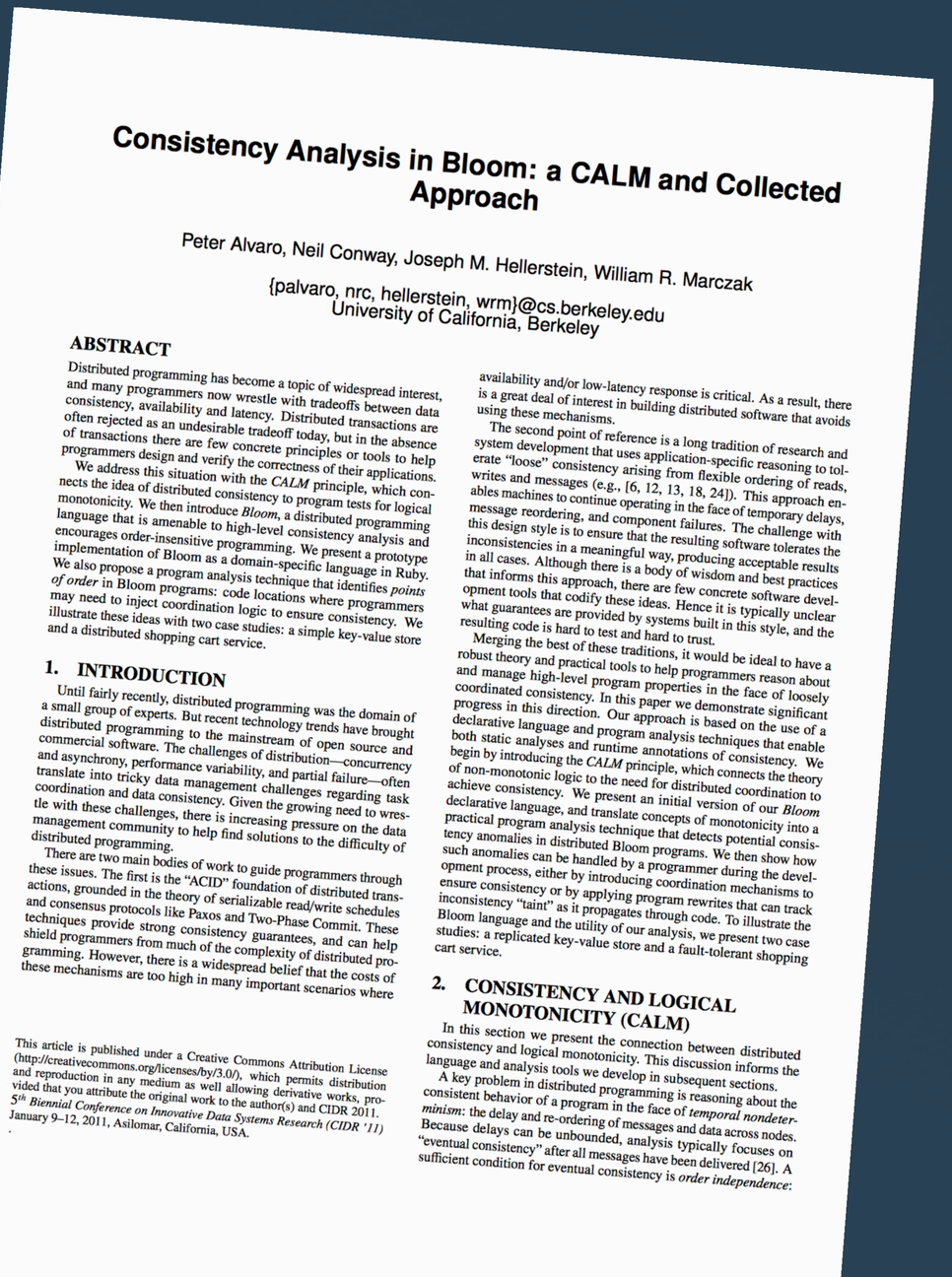
Using send and expect, the code for such a process would be:

```
ping :: ProcessM ()
ping = do Pong partner ← expect
          self ← getSelfPid
          send partner (Ping self)
          ping
```

The equivalent code in Erlang looks like this:

```
ping() → receive
           {pong, Partner} →
             Partner ! {ping, self()}
         end,
         ping().
```

BLOOM
(2011)



Consistency Analysis in Bloom: a CALM and Collected Approach

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak

{palvaro, nrc, hellerstein, wrm}@cs.berkeley.edu
University of California, Berkeley

ABSTRACT

Distributed programming has become a topic of widespread interest, and many programmers now wrestle with tradeoffs between data consistency, availability and latency. Distributed transactions are often rejected as an undesirable tradeoff today, but in the absence of transactions there are few concrete principles or tools to help programmers design and verify the correctness of their applications.

We address this situation with the *CALM* principle, which connects the idea of distributed consistency to program tests for logical monotonicity. We then introduce *Bloom*, a distributed programming language that is amenable to high-level consistency analysis and encourages order-insensitive programming. We present a prototype implementation of Bloom as a domain-specific language in Ruby. We also propose a program analysis technique that identifies *points of order* in Bloom programs: code locations where programmers may need to inject coordination logic to ensure consistency. We illustrate these ideas with two case studies: a simple key-value store and a distributed shopping cart service.

1. INTRODUCTION

Until fairly recently, distributed programming was the domain of a small group of experts. But recent technology trends have brought distributed programming to the mainstream of open source and commercial software. The challenges of distribution—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure on the data management community to help find solutions to the difficulty of distributed programming.

There are two main bodies of work to guide programmers through these issues. The first is the "ACID" foundation of distributed transactions, grounded in the theory of serializable read/write schedules and consensus protocols like Paxos and Two-Phase Commit. These techniques provide strong consistency guarantees, and can help shield programmers from much of the complexity of distributed programming. However, there is a widespread belief that the costs of these mechanisms are too high in many important scenarios where availability and/or low-latency response is critical. As a result, there is a great deal of interest in building distributed software that avoids using these mechanisms.

The second point of reference is a long tradition of research and system development that uses application-specific reasoning to tolerate "loose" consistency arising from flexible ordering of reads, writes and messages (e.g., [6, 12, 13, 18, 24]). This approach enables machines to continue operating in the face of temporary delays, message reordering, and component failures. The challenge with this design style is to ensure that the resulting software tolerates the inconsistencies in a meaningful way, producing acceptable results in all cases. Although there is a body of wisdom and best practices that informs this approach, there are few concrete software development tools that codify these ideas. Hence it is typically unclear what guarantees are provided by systems built in this style, and the resulting code is hard to test and hard to trust.

Merging the best of these traditions, it would be ideal to have a robust theory and practical tools to help programmers reason about and manage high-level program properties in the face of loosely coordinated consistency. In this paper we demonstrate significant progress in this direction. Our approach is based on the use of a declarative language and program analysis techniques that enable both static analyses and runtime annotations of consistency. We begin by introducing the *CALM* principle, which connects the theory of non-monotonic logic to the need for distributed coordination to achieve consistency. We present an initial version of our *Bloom* declarative language, and translate concepts of monotonicity into a practical program analysis technique that detects potential consistency anomalies in distributed Bloom programs. We then show how such anomalies can be handled by a programmer during the development process, either by introducing coordination mechanisms to ensure consistency or by applying program rewrites that can track inconsistency "taint" as it propagates through code. To illustrate the Bloom language and the utility of our analysis, we present two case studies: a replicated key-value store and a fault-tolerant shopping cart service.

2. CONSISTENCY AND LOGICAL MONOTONICITY (CALM)

In this section we present the connection between distributed consistency and logical monotonicity. This discussion informs the language and analysis tools we develop in subsequent sections.

A key problem in distributed programming is reasoning about the consistent behavior of a program in the face of *temporal nondeterminism*: the delay and re-ordering of messages and data across nodes. Because delays can be unbounded, analysis typically focuses on "eventual consistency" after all messages have been delivered [26]. A sufficient condition for eventual consistency is *order independence*:

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9–12, 2011, Asilomar, California, USA.

Why Bloom?

Why Bloom?

{palvaro, hfix, hellerstein, wrm}@cs.berkeley.edu
University of California, Berkeley

ABSTRACT

Distributed programming has become a topic of widespread interest, and many programmers now wrestle with tradeoffs between data consistency, availability and latency. Distributed transactions are often rejected as an undesirable tradeoff today, but in the absence of transactions there are few concrete principles or tools to help programmers design and verify the correctness of their applications.

We address this situation with the *CALM* principle, which connects the idea of distributed consistency to program tests for logical monotonicity. We then introduce *Bloom*, a distributed programming language that is amenable to high-level consistency analysis and encourages order-insensitive programming. We present a prototype implementation of Bloom as a domain-specific language in Ruby. We also propose a program analysis technique that identifies *points of order* in Bloom programs: code locations where programmers may need to inject coordination logic to ensure consistency. We illustrate these ideas with two case studies: a simple key-value store and a distributed shopping cart service.

1. INTRODUCTION

Until fairly recently, distributed programming was the domain of a small group of experts. But recent technology trends have brought distributed programming to the mainstream of open source and commercial software. The challenges of distribution—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure on the data management community to help find solutions to the difficulty of distributed programming.

There are two main bodies of work to guide programmers through these issues. The first is the "ACID" foundation of distributed transactions, grounded in the theory of serializable read/write schedules and consensus protocols like Paxos and Two-Phase Commit. These techniques provide strong consistency guarantees, and can help shield programmers from much of the complexity of distributed programming. However, there is a widespread belief that the costs of these mechanisms are too high in many important scenarios where availability and/or low-latency response is critical. As a result, there is a great deal of interest in building distributed software that avoids using these mechanisms.

The second point of reference is a long tradition of research and system development that uses application-specific reasoning to tolerate "loose" consistency arising from flexible ordering of reads, writes and messages (e.g., [6, 12, 13, 18, 24]). This approach enables machines to continue operating in the face of temporary delays, message reordering, and component failures. The challenge with this design style is to ensure that the resulting software tolerates the inconsistencies in a meaningful way, producing acceptable results in all cases. Although there is a body of wisdom and best practices that informs this approach, there are few concrete software development tools that codify these ideas. Hence it is typically unclear what guarantees are provided by systems built in this style, and the resulting code is hard to test and hard to trust.

Merging the best of these traditions, it would be ideal to have a robust theory and practical tools to help programmers reason about and manage high-level program properties in the face of loosely coordinated consistency. In this paper we demonstrate significant progress in this direction. Our approach is based on the use of a declarative language and program analysis techniques that enable both static analyses and runtime annotations of consistency. We begin by introducing the *CALM* principle, which connects the theory of non-monotonic logic to the need for distributed coordination to achieve consistency. We present an initial version of our *Bloom* declarative language, and translate concepts of monotonicity into a practical program analysis technique that detects potential consistency anomalies in distributed Bloom programs. We then show how such anomalies can be handled by a programmer during the development process, either by introducing coordination mechanisms to ensure consistency or by applying program rewrites that can track inconsistency "taint" as it propagates through code. To illustrate the Bloom language and the utility of our analysis, we present two case studies: a replicated key-value store and a fault-tolerant shopping cart service.

2. CONSISTENCY AND LOGICAL MONOTONICITY (CALM)

In this section we present the connection between distributed consistency and logical monotonicity. This discussion informs the language and analysis tools we develop in subsequent sections.

A key problem in distributed programming is reasoning about the consistent behavior of a program in the face of *temporal nondeterminism*: the delay and re-ordering of messages and data across nodes

5th Biennial Conference

Bloom is:

1. A declarative language.
2. Program analysis techniques which enable both static analysis and runtime annotations of consistency.

Bloom is:

1. A declarative language.
2. Program analysis techniques which enable both static analysis and runtime annotations of consistency.

But what is determined in this analysis?

Bloom is:

1. A declarative language.
2. Program analysis techniques which enable both static analysis and runtime annotations of consistency.

But what is determined in this analysis?

CALM Principle:

Tight relationship between Consistency And Logical Monotonicity.

Monotonic programs guarantee eventual consistency under any interleaving of delivery and computation.

By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously valid element of an output set—requires coordination schemes that “wait” until inputs can be guaranteed to be complete.

Bloom is:

1. A declarative language.
2. Program analysis techniques which enable both static analysis and runtime annotations of consistency.

But what is determined in this analysis?

(Syntactic) checks for monotonicity!

CALM Principle:

Tight relationship between Consistency And Logical Monotonicity.

Monotonic programs guarantee eventual consistency under any interleaving of delivery and computation.

By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously valid element of an output set—requires coordination schemes that “wait” until inputs can be guaranteed to be complete.

Overview of Bloom

No ordering.

Programs are bundles of declarative statements about collections of tuples.

Statements defined wrt “timesteps”

No mutable state. Side effect-free.

Overview of Bloom

No ordering.

Programs are bundles of declarative statements about collections of tuples.

Statements defined wrt "timesteps"

No mutable state. Side effect-free.

State in Bloom is managed using a series of collection objects and operations on those objects.

Type	Behavior
table	A collection whose contents persist across timesteps.
scratch	A collection whose contents persist for only one timestep.
channel	A scratch collection with one attribute designated as the *location specifier*. Tuples "appear" at the network address stored in their location specifier.
periodic	A scratch collection of key-value pairs (`id`, `timestamp`). The definition of a periodic collection is parameterized by a `period` in seconds; the runtime system arranges (in a best-effort manner) for tuples to "appear" in this collection approximately every `period` seconds, with a unique `id` and the current wall-clock time.
interface	A scratch collection specially designated as an interface point between modules.

Op	Valid lhs types	Meaning
=	**scratch**	rhs defines the contents of the lhs for the current timestep. lhs must not appear in lhs of any other statement.
<=	**table, scratch**	lhs includes the content of the rhs in the current timestep.
<+	**table, scratch**	lhs will include the content of the rhs in the next timestep.
<-	**table**	tuples in the rhs will be absent from the lhs at the start of the next timestep.
<~	**channel**	tuples in the rhs will appear in the (remote) lhs at some non-deterministic future time.

Figure 1: Bloom collection types and operators.

Overview of Bloom

Also comes with a number of familiar and useful collection methods:

Method	Description
`bc.map`	Takes a code block and returns the collection formed by applying the code block to each element of bc.
`bc.flat_map`	Equivalent to `map`, except that any nested collections in the result are flattened.
`bc.reduce`	Takes a `memo` variable and code block, and applies the block to `memo` and each element of bc in turn.
`bc.empty?`	Returns true if bc is empty.
`bc.include?`	Takes an object and returns true if that object is equal to any element of bc.
`bc.group`	Takes a list of grouping columns, a list of aggregate expressions and a code block. For each group, computes the aggregates and then applies the code block to the group/aggregation result.
`join`, `leftjoin`, `outerjoin`, `natjoin`	Methods of the Bud class to compute join variants over BudCollections. `join`, `leftjoin` and `outerjoin` take an array of collections to join, as well as a variable-length list of arrays of join conditions. The natural join `natjoin` takes only the array of BudCollection objects as an argument.

Figure 3: Commonly used methods of the BudCollection class.

Statements are in the form of:

```
<collection> <operation> <collection-expression>
```

like map or join

For example,

```
kvstate <+ kvput.map{|p| [p.key, p.value]}
```

Statements are specified on modules.

Order of statements doesn't matter.

How is Bloom implemented?

In pure Ruby!

A Bloom program is just a Ruby class definition.

> Because Bud is pure Ruby, some programmers may choose to embed it as a domain-specific language (DSL) within traditional imperative Ruby code. In fact, nothing prevents a subclass of Bud from having both Bloom code in `declare` methods and imperative code in traditional Ruby methods. This is a fairly common usage model for many DSLs. A mixture of declarative Bloom methods and imperative Ruby allows the full range of existing Ruby code—including the extensive RubyGems repositories—to be combined with checkable distributed Bloom programs. The analyses we describe in the remaining sections still apply in these cases; the imperative Ruby code interacts with the Bloom logic in the same way as any external agent sending and receiving network messages.

(Considered to be a benefit.)

LASP
(2015)

Lasp: A Language for Distributed, Coordination-Free Programming

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Abstract

We propose Lasp, a new programming model designed to simplify large-scale distributed programming. Lasp combines ideas from deterministic dataflow programming together with conflict-free replicated data types (CRDTs). This provides support for computations where not all participants are online together at a given moment. The initial design presented here provides powerful primitives for composing CRDTs, which lets us write long-lived fault-tolerant distributed applications with nonmonotonic behavior in a monotonic framework. Given reasonable models of node-to-node communications and node failures, we prove formally that a Lasp program can be considered as a functional program that supports functional reasoning and programming techniques. We have implemented Lasp as an Erlang library built on top of the Riak Core distributed systems framework. We have developed one nontrivial large-scale application, the advertisement counter scenario from the SyncFree research project. We plan to extend our current prototype into a general-purpose language in which synchronization is used as little as possible.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Eventual Consistency, Commutative Operations, Erlang

1. Introduction

Synchronization of data across systems is becoming increasingly expensive and impractical when running at the scale required by "Internet of Things" [29] applications and large online mobile games.[1] Not only does the time required to coordinate with an ever growing number of clients increase with each additional client, but techniques that rely on coordination of shared state, such as Paxos and state-machine replication, grow in complexity with partial replication, dynamic membership, and unreliable networks. [14]

This is further complicated by an additional requirement for both of these applications: each must tolerate periods without connectivity while allowing local copies of replicated state to change. For example, mobile games should allow players to continue to accumulate achievements or edit their profile while they are riding in the subway without connectivity; "Internet of Things" applications should be able to aggregate statistics from a power meter during a snowstorm when connectivity is not available, and later synchronize when connectivity is restored. Because of these requirements, the burden is placed on the programmer of these applications to ensure that concurrent operations performed on replicated data have both a deterministic and desirable outcome.

For example, consider the case where a user's gaming profile is replicated between two mobile devices. Concurrent operations, which can be thought of as operations performed during the period where both clients are online but without communication, can modify the same state: the burden is placed on the application developer to write application logic that resolves these conflicting updates. This is true even if the changes commute: for instance, concurrent modifications to the user profile where client A modifies the profile photo and client B modifies the profile's e-mail address.

Recently, a formalism has been proposed by Shapiro et al. for supporting deterministic resolution of individual objects that are acted upon concurrently in a distributed system. These data types, referred to as Conflict-Free Replicated Data Types (CRDTs), provide a property formalized as Strong Eventual Consistency: given all updates to an object are eventually delivered in a distributed system, all copies of that object will converge to the same state. [32, 33]

Strong Eventual Consistency (SEC) results in deterministic resolution of concurrent updates to replicated state. This property is highly desirable in a distributed system because it no longer places the resolution logic in the hands of the programmer; programmers are able to use replicated data types that function as if they were their sequential counterparts. However, it has been shown that arbitrary composition of these data types is nontrivial. [6, 13, 15, 26]

To achieve this goal, we propose a novel programming model aimed at simplifying correct, large-scale, distributed programming, called Lasp.[2] This model provides the ability to use operations from functional programming to deterministically compose CRDTs into larger computations that observe the SEC property; these applications support programming with data structures whose values appear nonmonotonic externally, while computing internally with the objects' monotonic metadata. This model builds on our previous work, Derflow and Derflow$_L$ [12, 27], which provide a distributed, fault-tolerant variable store powering a deterministic concurrency programming model.

This paper has the following contributions:

- **Formal semantics:** We provide the formal semantics for Lasp: the monotonic **read** operation; functional programming operations over sets, including **map**, **filter**, and **fold**; and set-theoretic operations, including **product**, **union**, and **intersection**.
- **Formal theorem:** We formally prove that a distributed execution of a Lasp program can be considered a functional pro-

[1] Rovio, developer of the popular "Angry Birds" game franchise reported that during the month of December 2012 they had 263 million active users. This does not account for users who play the game on multiple devices, which is an even larger number of devices requiring some form of shared state in the form of statistics, metrics, or leaderboards. [3]

[2] Inspired by LISP's etymology of "LISt Proce... structure is a join-semilattice, ho...

Why Lasp?

It's the future. It's 2015.
Everything is distributed somehow.

- mobile games (online/offline)
- mobile apps (online/offline)
- "internet of s...things..."
- ad counters (everyones' favorite)

Synchronization is a ridiculous idea when one app has to make calls to 3 dozen services during usage.

Also, state *has to* be replicated sometimes.
e.g., online rpg game. health indicators during a sword fight.

Why Lasp?

It's the future. It's 2015.
Everything is distributed somehow.

- mobile games (online/offline)
- mobile apps (online/offline)
- "internet of s...things..."
- ad counters (everyones' favorite)

Synchronization is a ridiculous idea when one app has to make calls to 3 dozen services during usage.

Also, state *has to* be replicated sometimes.
e.g., online rpg game. health indicators during a sword fight.

CRDTs

Lasp builds CRDTs into the language.

Why Lasp?

It's the future. It's 2015.
Everything is distributed somehow.

- mobile games (online/offline)
- mobile apps (online/offline)
- "internet of s...things..."
- ad counters (everyones' favorite)

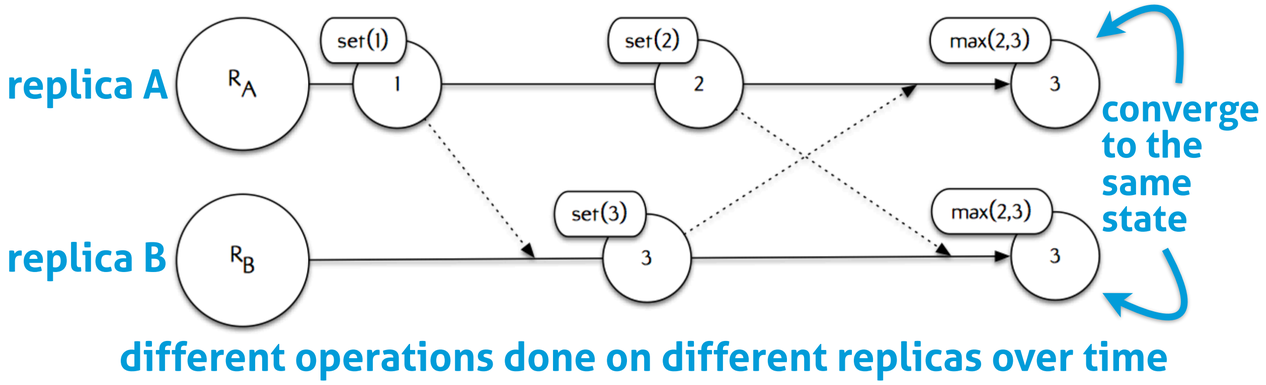
Synchronization is a ridiculous idea when one app has to make calls to 3 dozen services during usage.

Also, state *has to* be replicated sometimes.
e.g., online rpg game. health indicators during a sword fight.

CRDTs

Lasp builds CRDTs into the language.

BASIC IDEA:
Data type that can be replicated, & can deal with temporary divergence at each replica.
Eventually consistent.



different operations done on different replicas over time

Why Lasp?

It's the future. It's 2015.
Everything is distributed somehow.

- mobile games (online/offline)
- mobile apps (online/offline)
- "internet of s...things..."
- ad counters (everyones' favorite)

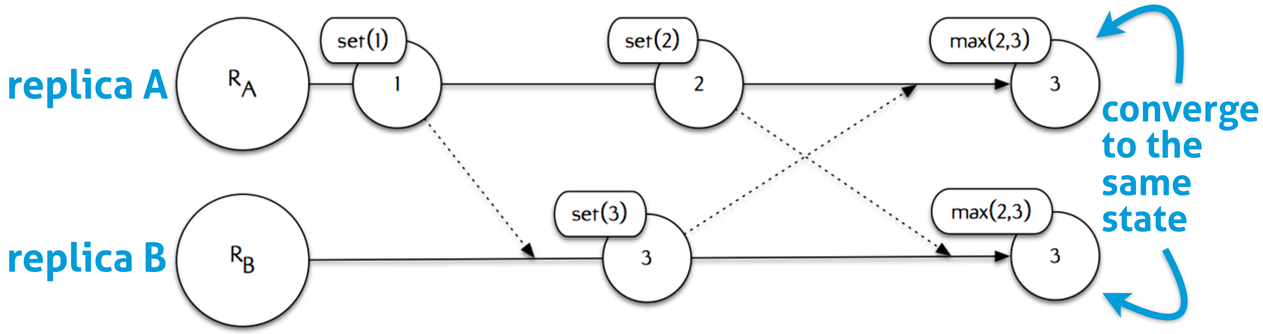
Synchronization is a ridiculous idea when one app has to make calls to 3 dozen services during usage.

Also, state *has to* be replicated sometimes.
e.g., online rpg game. health indicators during a sword fight.

CRDTs

Lasp builds CRDTs into the language.

BASIC IDEA:
Data type that can be replicated, & can deal with temporary divergence at each replica.
Eventually consistent.



different operations done on different replicas over time

Many kinds of data structures can be realized as a CRDT. E.g., sets, counters, registers, dictionaries, graphs, ...

Lasp Overview

BIG IDEA:
Let's build convergent computations
by **♥composing♥** CRDTs.

Primary data type in Lasp:
CRDT

3 APIs.
Core, Functional, Set-Theoretic.

Core API:

responsible for defining variables, setting their values and reading the result of variable assignments.

`declare, bind, update:` *do what you expect*

`read(x, v):` ***monotonic read operation***

`strict_read(x, v):` ***monotonic read operation, waits till x is > v***

Lasp Overview

BIG IDEA:
Let's build convergent computations by **♥composing♥** CRDTs.

Primary data type in Lasp:
CRDT

3 APIs.
Core, Functional, Set-Theoretic.

Core API:

responsible for defining variables, setting their val
and reading the result of variable assignments.

`declare, bind, update:` *do what you expect*

`read(x, v):` ***monotonic read operation***

`strict_read(x, v):` ***monotonic read operation,***
waits till x is > v

Functional API

define processes that never terminate; each process is responsible for reading subsequent values of the input and writing to the output.

> `map(x, f, y)`:
> Apply function f over x into y.
>
> `filter (x, p, y)`:
> Apply filter predicate p over x into y.
>
> `fold (x, op, y)`:
> Fold values from x into y using operation op.

Set-Theoretic API

define processes that never terminate; each process is responsible for reading subsequent values of the input and writing to the output.

> `product(x, y, z)`:
>
> `union(x, y, z)`:
> `intersection(x, y, z)`:

Lasp Quick Glimpse

```
1  %% Create initial set S1.
2  {ok, S1} = lasp:declare(riak_dt_orset),
3
4  %% Add elements to initial set S1 and update.
5  {ok, _} = lasp:update(S1, {add_all, [1,2,3]}, a),
6
7  %% Create second set S2.
8  {ok, S2} = lasp:declare(riak_dt_orset),
9
10 %% Apply map operation between S1 and S2.
11 {ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Figure 4: *Map function applied to an OR-Set using the Erlang API of our Lasp prototype. We ignore the return values of the functions, given the brevity of the example.*

How is Lasp implemented?

as an Erlang library!

Built on top of the riak_dt library

(which provides state-based CRDTs in Erlang)

5.1 Distribution

Lasp distributes data using the Riak Core distributed systems framework [22], which is based on the Dynamo system [18].

Riak Core The Riak Core library provides a framework for building applications in the style of the original Dynamo system. Riak Core provides library functions for cluster management, dynamic membership and failure detection.

But where did all of the distributed languages go?

Nowhere.

They're just distributed DSLs now.

Or libraries.

Whatever you want to call them.

At some point, you've got to ask yourself, what the hell is a language?

Nowhere.

DSL approach actually the most sound, if you can express your programming model within some general-purpose language.

Hey, consistency is important!
Also, dealing with partial failure.
Also RPC, promise pipelining.

Argus

Object model should be unified.
Objects should be mobile!
And they should be efficient!

Emerald

Message-passing is king.
Processes should be many & light.
& totally independent too.

Erlang

Guys, message-passing is cray. Shared-memory model is more natural. Tuplespaces = efficient!

Linda

We can embed Erlang in Scala. We don't need a VM for lightweight processes! Library FTW!

Scala Actors

Message-passing would be nice in Haskell. And we can do it better! Types and monads FTW!

Cloud Haskell

Guys, guys, consistency is important. Things should be monotonic. Datalog. Disorderly.

Bloom

Consistency is important, but so is composability. Let’s put it in a language!

Lasp

Fin. + References!

Liskov, B. & Scheifler, R. (1983). Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM Transactions on Programming Languages and Systems. 5(3), pp. 381-404.

Black, A., Hutchinson, N., Jul, E., Levy, H., & Carter, L. (1987). Distribution and Abstract Types in Emerald. IEEE Transactions on Software Engineering,1, 65–76.

Liskov, B., Curtis, D., Johnson, P., & Scheifer, R. (1987). Implementation of Argus. Proceedings of the 11th ACM Symposium on Operating Systems Principles. pp. 111-122.

Liskov, B. (1988). Distributed Programming in Argus.Communications of the ACM,31(3), 300–312.

Liskov, B., & Shrira, L. (1988). Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. SIGPLAN Not.,23(7), 260–267.

Carriero, N., Gelernter, D., Mattson, T., & Sherman, A. (1994). The Linda Alternative to Message-Passing Systems. Journal of Parallel Computing., 20(4), pp. 633-655.

Black, A., Hutchinson, N., Jul, E., & Levy, H. (2007). The Development of the Emerald Programming Language. Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages, 11–11. ACM.

Haller, P., & Odersky, M. (2009). Scala Actors: Unifying Thread-based and Event-based Programming.Theor. Comput. Sci.,410(2-3), 202–220.

Armstrong, J. (2010). Erlang.Communications of the ACM,53(9), 68–75.

Alvaro, P., Conway, N., & Hellerstein, J. (2011). Consistency Analysis in Bloom: a CALM and Collected Approach. Proceedings of the Conference on Innovative Data Systems Research.

Epstein, J., Black, A., & Peyton-Jones, S. (2011). Towards Haskell in the Cloud. 4th ACM Symposium on Haskell, 2011 ACM SIGPLAN pp. 118-129.

Meiklejohn, C. (2015). Lasp: A Language for Distributed, Coordination-Free Programming. 17th International Symposium on Principles and Practice of Declarative Programming.