Distributed Programming in Argus

Barbara Liskov
Caitie McCaffrey
Distributed Systems Engineer

@caitie
CaitieM.com
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.

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 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
“Technological advances have made it cost effective to construct large systems from collections of computer connected via networks. To support such systems, there is a growing need for effective way to organize and maintain distributed programs”
Remote Procedure Calls

“We believe that the most desirable form of communication is the paired send and reply”
Remote Procedure Calls

“We believe the form of communication that is needed is remote procedure call with at-most-once semantics”
RPC In Argus
Guardians

“a special kind of abstract object whose purpose is to encapsulate a resource”
Handlers

“it permits its resource to be accessed by means of special procedures, called handlers”
Banking with Argus
Branch Guardian

Handlers

Open (a: account_number)
Close (a: account_number)
Deposit (a: account_number, amt: int)
Withdraw (a: account_number, amt: int)
Total()
Branch A

Account 123: $100
Account 456: $50

Open (123)
Deposit(123, 100)
Open (456)
Deposit(456, 50)

Branch B

Account 789: $250

Open (789)
Deposit(789, 250)
RPC & Atomicity
"An adequate language must provide a modular, reasonably automatic method for achieving consistency."

Distributed Programming in Argus
Atomicity

“Our solution to the problem of maintaining consistent distributed data in the face of concurrent, potentially interfering activities, and in the face of system failures such as node crashes and network disruptions while these activities are running is to make activities atomic”
Atomicity
In Argus
Actions

Serializable

Total (abort or commit)
Atomic Objects

Atomic Abstract Data Type

Argus Provides: atomic arrays, records, variants, characters, and integers
Locking Rules

Read Write Lock

Multiple Readers are Allowed

Readers Exclude Writers

Writer Exclude other Writers & Readers
“Computation in Argus starts as a top action at some guardian. The computation spreads to other guardians by means of handler calls. Execution of a handler call may cause some objects at the handler’s guardian to be modified, and may in turn lead to further calls”
Banking with Argus
Transfer( amt: int,
from: account_number, to: account_number)
Transfer Action

SubAction: Deposit (123, 50)
SubAction: Withdraw(789, 50)

Branch A
Account 123 : $100
Account 456: $50

Branch B
Account 789: $250
enter topaction
  coenter
    action
      branchA.Deposit(123, 50)
    action
      branchB.Withdraw(789, 50)
  end
end
Transfer Action

Branch A
- Account 123: $100
- Account 456: $50

Branch B
- Account 789: $250

Deposit (123, 50) → Withdraw (789, 50)
Deposit (123, 50)

Branch A

Account 123: $150
Account 123: $100
Account 456: $50

Branch B

Account 789: $200
Account 789: $250

Withdraw (789, 50)
Transfer Action

Branch A
- Account 123: $150
- Account 123: $100
- Account 456: $50

Branch B
- Account 789: $200
- Account 789: $250

Deposit Success
Withdraw Success
Transfer Action

Commit Top Level Action
2PC

Branch A
V1 Account 123 : $150
Account 123 : $100
Account 456: $50

Branch B
V1 Account 789: $200
Account 789: $250
Transfer Action

Commit Top Level Action
2PC: Phase 1

Prepare

Branch A

Account 123: $150
Account 123: $100
Account 456: $50

Prepare

Branch B

Account 789: $200
Account 789: $250
Account 123: $100
Transfer Action

Commit Top Level Action
2PC: Phase 1

Prepare
Success

Branch A

Account 123: $150
Account 123: $100
Account 456: $50

Prepare
Success

Branch B

Account 789: $200
Account 789: $250

Account 123: $150
Account 789: $200
Account 456: $50
Account 789: $250
Transfer Action

Commit Top Level Action

2PC : Phase 2

Branch A

Account 123 : $150
Account 456: $50

Branch B

Account 789: $200
Transfer Action

Commit Top Level Action
2PC : Complete

Branch A
Account 123 : $150
Account 456: $50

Branch B
Account 789: $200
Banking with Argus
**Transfer Action**

SubAction: Deposit (123, 100)
SubAction: Withdraw(456, 100)

**Branch A**
- Account 123: $150
- Account 456: $50

**Branch B**
- Account 789: $200
Deposit (123, 100)
Withdraw(456, 100)

Branch A
- Account 123: $150
- Account 456: $50

Branch B
- Account 789: $200
Deposit Success
Withdraw Abort

**Branch A**
- V1: Account 123: $250
- Account 123: $150
- Account 456: $50

**Branch B**
- Account 789: $200
Transfer Action

Abort Top Level Action
2PC : Abort

Abort Deposit

Branch A

Account 123 : $250
Account 123 : $150
Account 456: $50

Branch B

Account 789: $200
Transfer Action

Abort Top Level Action
2PC : Abort

Deposit
Aborted

Branch A
Account 123 : $150
Account 456: $50

Branch B
Account 789: $200
Transfer Action ✗

Branch A
- Account 123: $150
- Account 456: $50

Branch B
- Account 789: $200
Sub-Actions

“Argus allows actions to be nested; thus an action can have one or more sub-actions”

Distributed Programming in Argus
Sub-Actions

- Replicate Action
  - Replica A
  - Replica B
Sub-Actions

Transfer

Deposit

Replicate
Republic Deposit

Replica A
Replica B

Withdraw

Replicate
Withdraw

Replica A
Replica B
Sub-Actions

- Transfer
  - Deposit
    - Replicate
      - Deposit
        - Replica A
        - Replica B
    - Withdraw
      - Replicate
        - Withdraw
          - Replica A
          - Replica B

Sub-Actions

- Replicate
- Deposit
- Replica A
- Replica B

- Transfer
  - Deposit
  - Withdraw
    - Replicate
      - Withdraw
        - Replica A
        - Replica B

- Abort
Sub-Actions

- Transfer
  - Deposit
    - Replicate Deposit
      - Replica A
      - Replica B
    - Withdraw
      - Replicate Withdraw
        - Replica A
        - Replica B
Sub-Actions

Transfer
- Deposit
  - Replicate Deposit
    - Replica A
    - Replica B
- Withdraw
  - Replicate Withdraw
    - Replica A
    - Replica B
Sub-Actions

Transfer

Deposit
- Replicate Deposit
  - Replica A
  - Replica B

Withdraw
- Replicate Withdraw
  - Replica A
  - Replica B

2PC: Abort
Sub-Actions

“Argus runs every handler call as a sub-action...this extra action ensures that calls have a zero or one semantics.”

Distributed Programming in Argus
What if Sub-Actions try to access the same atomic object?
What if Sub-Actions try to access the same atomic object?

Locking Rules for Sub-Actions

**Read Lock:** All holders of write locks on x must be ancestors of S.

**Write Lock:** All holders of write locks on x must be ancestors of S.
Version Management Rules for Sub Actions

**Commit:** S’s parent acquires S’s lock on x. If S holds a write lock on x, then S’s version becomes S’s parent version

**Abort:** S’s lock and version (if any) are discarded

Distributed Programming in Argus
Sub Action Locking

Account Balance Object

1.5% Interest

2% Interest

Read Balance

Value

$200

Locked
Sub Action Locking

1.5% Interest

2% Interest

Read Balance

Account Balance Object

Value

$203
$200

X: Write Lock (V1)
Sub Action Locking

X: Write Lock (V1)
Y: Write Lock (V2)
Z: Read Balance

Account Balance Object

Value
- V2: $207.06
- V1: $203
- X: Write Lock (V1)
- Y: Write Lock (V2)

1.5% Interest
2% Interest

Read Balance
Sub Action Locking

Account Balance Object

Value

- V2: $207.06
- V1: $203
- $200

X: Write Lock (V1)
Y: Write Lock (V2)
Z: Read Lock (V2)
W: .5% Interest
X: .5% Interest
Y: 2% Interest
Z: Read Lock (V2)
W: 1.5% Interest

Z: Read Lock (V2)
Y: Write Lock (V2)
X: Write Lock (V1)
Sub Action Locking

Account Balance Object

<table>
<thead>
<tr>
<th>Value</th>
<th>$208.10</th>
<th>$207.06</th>
<th>$203</th>
<th>$200</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

W: Write Lock (V3)
Z: Read Lock (V2)
Y: Write Lock (V2)
X: Write Lock (V1)

X: 1.5% Interest
Y: 2% Interest
Z: Read Balance
W: .5% Interest
Sub Action Locking

Account Balance Object

<table>
<thead>
<tr>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>$203</td>
</tr>
<tr>
<td>V2</td>
<td>$207.06</td>
</tr>
<tr>
<td>V3</td>
<td>$208.10</td>
</tr>
</tbody>
</table>

X: Write Lock (V1)

Y: Write Lock (V2)

Z: Write Lock (V3)

1.5% Interest

2% Interest

.5% Interest

Read Balance

W

Interest

1.5%

2%

.5%
Sub Action Locking

X: Write Lock (V1)

1.5% Interest

Y: Write Lock (V3)

2% Interest

Read Balance

Z

.5% Interest

Account Balance Object

Value

<table>
<thead>
<tr>
<th>V3</th>
<th>$208.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td>$207.06</td>
</tr>
<tr>
<td>V1</td>
<td>$203</td>
</tr>
</tbody>
</table>

W

$200

Y: Write Lock (V3)

X: Write Lock (V1)
Sub Action Locking

X: Write Lock (V3)

Account Balance Object

Value
- V3: $208.10
- V2: $207.06
- V1: $203
- $200

1.5% Interest
2% Interest
.5% Interest

Read Balance

X

Y

Z

W
Sub Action Locking

Account Balance Object

<table>
<thead>
<tr>
<th>X</th>
<th>Write Lock (V3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>$200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
<th>Write Lock (V3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3</td>
<td>$208.10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z</th>
<th>Write Lock (V3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3</td>
<td>$207.06</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
<th>Write Lock (V3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3</td>
<td>$200</td>
</tr>
</tbody>
</table>

1.5% Interest

Read Balance

0.5% Interest

Two Phase Commit
Sub Action Locking

Account Balance Object

X
1.5% Interest

Y
2% Interest

Z
Read Balance

W
.5% Interest

Value

$208.10
Problems with Argus
Deadlocks

“The concurrency that is built in to the mail system can lead to a number of deadlock situations”

Guardians and Actions: Linguistic Support for Robust, Distributed Programs

“As implemented most of the handlers can deadlock with other concurrent operations”

Distributed Programming in Argus
Deadlocks

“The programmer must think about deadlocks and starvation and implement the code to avoid them when possible”

Distributed Programming in Argus
Blocking Calls

“A new process is created to perform an incoming handler call ...(so guardians can) have the ability to execute many request concurrently ... if the guardian is running on a single-processor node, then only one process will be running at a time”
Promises:

linguistic support for efficient asynchronous procedure calls in distributed systems

B. Liskov, L. Shrira
So where do we go from here?
Guardians allow programs to be decomposed into units of tightly coupled data and processing.

Distributed Programming in Argus
Microservices
“Atomic actions are an important tool both for understanding what a system should do and for implementing it correctly”
Orleans: Distributed Virtual Actors for Programmability and Scalability

Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, Jorgen Thelin
Microsoft Research

Abstract

High-scale interactive services demand high throughput with low latency and high availability, difficult goals to meet with the traditional stateless 3-tier architecture. The actor model makes it natural to build a stateful middle tier and achieve the required performance. However, the popular actor model platforms still pass many distributed systems problems to the developers.

The Orleans programming model introduces the novel abstraction of virtual actors that solves a number of the complex distributed systems problems, such as reliability and distributed resource management, liberating the developers from dealing with those concerns. At the same time, the Orleans runtime enables applications to attain high performance, reliability and scalability.

This paper presents the design principles behind Orleans and demonstrates how Orleans achieves a simple programming model that meets these goals. We describe how Orleans simplified the development of several scalable production applications on Windows Azure and report on the performance of those applications compared to a more mature actor based platform. Orleans provides the required application-level semantics and consistency on a cache with fast response for interactive access.

The actor model offers an appealing solution to these challenges by relying on the function shipping paradigm. Actors allow building a stateful middle tier that has the performance benefits of a cache with data locality and the semantic and consistency benefits of encapsulated entities via application-specific operations. In addition, actors make it easy to implement horizontal, “social”, relations between entities in the middle tier.

Another view of distributed systems programmability is through the lens of the object-oriented programming (OOP) paradigm. While OOP is an intuitive way to model complex systems, it has been marginalized by the popular service-oriented architecture (SOA). One can still benefit from OOP when implementing service components. However, at the system level, developers have to think in terms of loosely-coupled partitioned services, which often do not match the application’s conceptual objects. This has contributed to the difficulty of building distributed systems by mainstream developers. The actor model brings OOP back to the client and the server to simplify development.
Halo 4: Statistics Service
Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity

Peter Bailis, Alan Fekete†, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and †University of Sydney

ABSTRACT
The rise of data-intensive “Web 2.0” Internet services has led to a range of popular new programming frameworks that collectively embody the latest incarnation of the vision of Object-Relational Mapping (ORM) systems, albeit at unprecedented scale. In this work, we empirically investigate modern ORM-backed applications’ use and disuse of database concurrency control mechanisms. Specifically, we focus our study on the common use of feral, or application-level, mechanisms for maintaining database integrity, which, across a range of ORM systems, often take the form of declarative correctness criteria, or invariants. We quantitatively analyze the use of these mechanisms in a range of open source applications written using the Ruby on Rails ORM and find that feral invariants are the most popular means of ensuring integrity (and, by usage, are over 37 times more popular than transactions). We evaluate which of these feral invariants actually ensure integrity (by usage, up to 86.9%) and which—due to concurrency errors and lack of database support—may lead to data corruption (the remainder), which we experimentally quantify. In light of these findings, we present recommendations for database system designers for better supporting these modern ORM programming patterns, thus eliminating their adverse effects on application integrity.

Rails is interesting for at least two reasons. First, it continues to be a popular means of developing responsive web application front-end and business logic, with an active open source community and user base. Rails recently celebrated its tenth anniversary and enjoys considerable commercial interest, both in terms of deployment and the availability of hosted “cloud” environments such as Heroku. Thus, Rails programmers represent a large class of consumers of database technology. Second, and perhaps more importantly, Rails is “opinionated software” [41]. That is, Rails embodies the strong personal convictions of its developer community, and, in particular, David Heinemeier Hansson (known as DHH), its creator. Rails is particularly opinionated towards the database systems that it tasks with data storage. To quote DHH:

“I don’t want my database to be clever! … I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you’ll have to try that logic from my dead, cold object-oriented hands … I want a single layer of cleverness: My domain model.” [55]

Thus, this wildly successful software framework bears an actively antagonistic relationship to database management systems, echoing
“We focus our study on the common use of feral or application-level, mechanisms for maintaining database integrity”
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

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 certainty is
CRDT in practice

* Stolen from Chris Meiklejohn
Spanner: Google’s Globally-Distributed Database


Google, Inc.

Abstract

Spanner is Google’s scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

Spanner’s main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like
“Spanner is the first system to distribute data at global scale and support externally-consistent distributed transaction”
'Cause I’m Strong Enough: Reasoning about Consistency Choices in Distributed Systems

Alexey Gotsman
IMDEA Software Institute, Spain

Hongseok Yang
University of Oxford, UK

Carla Ferreira
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa, Portugal

Mahsa Najafzadeh
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

Marc Shapiro
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

Abstract

Large-scale distributed systems often rely on replicated databases that allow a programmer to request different data consistency guarantees for different operations, and thereby control their performance. Using such databases is far from trivial: requesting stronger consistency in too many places may hurt performance, and requesting it in too few places may violate correctness. To help programmers in this task, we propose the first proof rule for establishing that a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure the preservation of a given data integrity invariant. Our rule is modular: it allows reasoning about the behaviour of every operation separately under some assumption on the behaviour of other operations. This leads to simple reasoning, which we have automated in an SMT-based tool. We present a nontrivial proof of soundness of our rule and illustrate its use on several examples.
public class Account extends AnnotatedSchema {

    @XPR(value = {"Int amount", "Int balance" }, type = XPR.Type.INSTANCE)
    @XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
    @Op(Account.Deposit.class)
    @Op(Account.Debit.class)

    public static class Deposit extends AnnotatedOperation {
        // Implementation for Deposit
    }

    @XPR(value = {"Int amount", "Int balance" }, type = XPR.Type.INSTANCE)
    @XPR(value = "amount >= 0", type = XPR.Type.PRECONDITION)
    @XPR(value = "balance := balance + amount", type = XPR.Type.EFFECT)

    public static class Debit extends AnnotatedOperation {
        // Implementation for Debit
    }
}
Conclusion
Thank You!

@caitie