



LVars:

**Lattice-based Data Structures
for Deterministic Parallelism**

Lindsey Kuper
Indiana University

RICON West, San Francisco, CA, USA
October 29, 2013

Jeff Dean



Google Fellow
Google, Inc.

The Tail at Scale: Achieving Rapid Response Times in Large Online Services

Peter Bailis



PhD Student
UC Berkeley

Bad As I Wanna Be: Coordination and Consistency in Distributed Databases

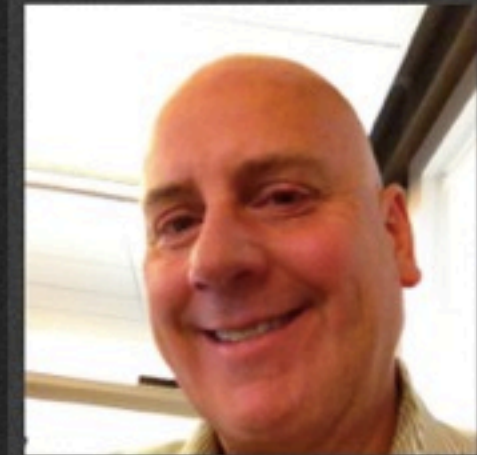
Justin Sheehy



Chief Technology Officer
Basho Technologies

Maximum Viable Product

Pat Helland



Architect
Salesforce.com

Keystone: Between a ROC and a SOFT Place

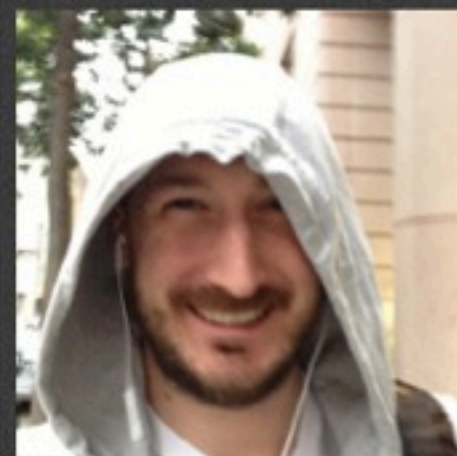
Diego Ongaro



PhD Student
Stanford University

The Raft Consensus Algorithm

Jeff Hodges



Distributed Systems Engineer
Twitter

Practicalities of Productionizing Distributed Systems

Lindsey Kuper



PhD Student
Indiana University

LVars: lattice-based data structures for deterministic parallelism

Michael Bernstein



Instigator
Code Climate

Distributed Systems Archaeology

Jeff Dean



Google Fellow
Google, Inc.

The Tail at Scale: Achieving Rapid Response Times in Large Online Services

Peter Bailis



PhD Student
UC Berkeley

Bad As I Wanna Be: Coordination and Consistency in Distributed Databases

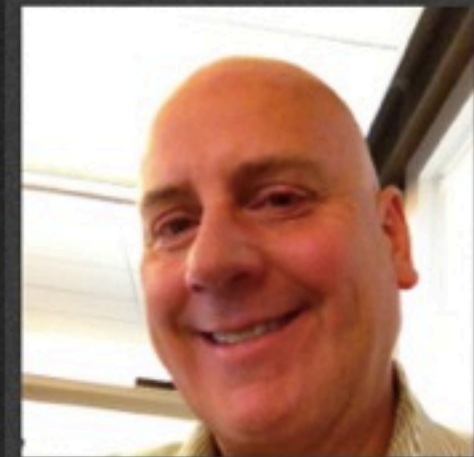
Justin Sheehy



Chief Technology Officer
Basho Technologies

Maximum Viable Product

Pat Helland



Architect
Salesforce.com

Keystone: Between a ROC and a SOFT Place

grad student

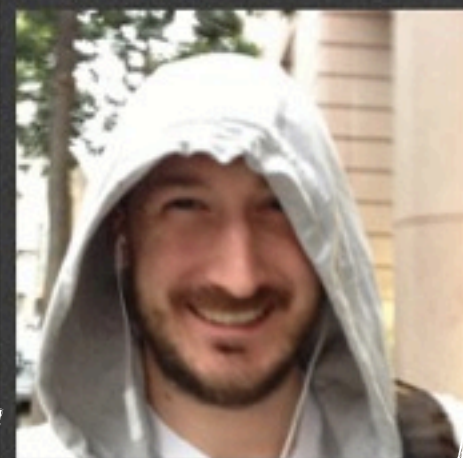
Diego Ongaro



PhD Student
Stanford University

The Raft Consensus Algorithm

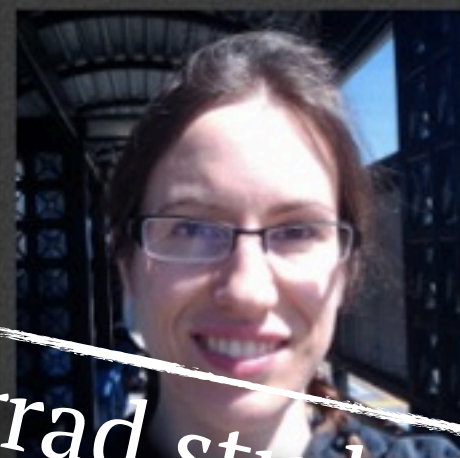
Jeff Hodges



Distributed Systems Engineer
Twitter

Practicalities of Productionizing Distributed Systems

Lindsey Kuper



PhD Student
Indiana University

LVars: lattice-based data structures for deterministic parallelism

Michael Bernstein



Instigator
Code Climate

Distributed Systems Archaeology

grad student

grad student

Jeff Dean



person with
real job

Google Fellow
Google, Inc.

*The Tail at Scale: Achieving Rapid
Response Times in Large Online
Services*

Peter Bailis



grad student

PhD Student
UC Berkeley

*Bad As I Wanna Be: Coordination
and Consistency in Distributed
Databases*

Justin Sheehy

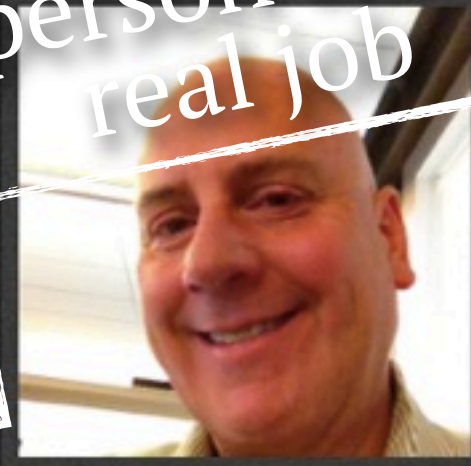


person with
real job

Chief Technology Officer
Basho Technologies

Maximum Viable Product

Pat Helland



person with
real job

Architect
Salesforce.com

*Keystone: Between a ROC and a
SOFT Place*

Diego Ongaro



grad student

PhD Student
Stanford University

The Raft Consensus Algorithm

Jeff Hodges



person with
real job

Distributed Systems Engineer
Twitter

*Practicalities of Productionizing
Distributed Systems*

Lindsey Kuper



grad student

PhD Student
Indiana University

*LVars: lattice-based data
structures for deterministic
parallelism*

Michael Bernstein



person with
real job

Investigator
Code Climate

Distributed Systems Archaeology

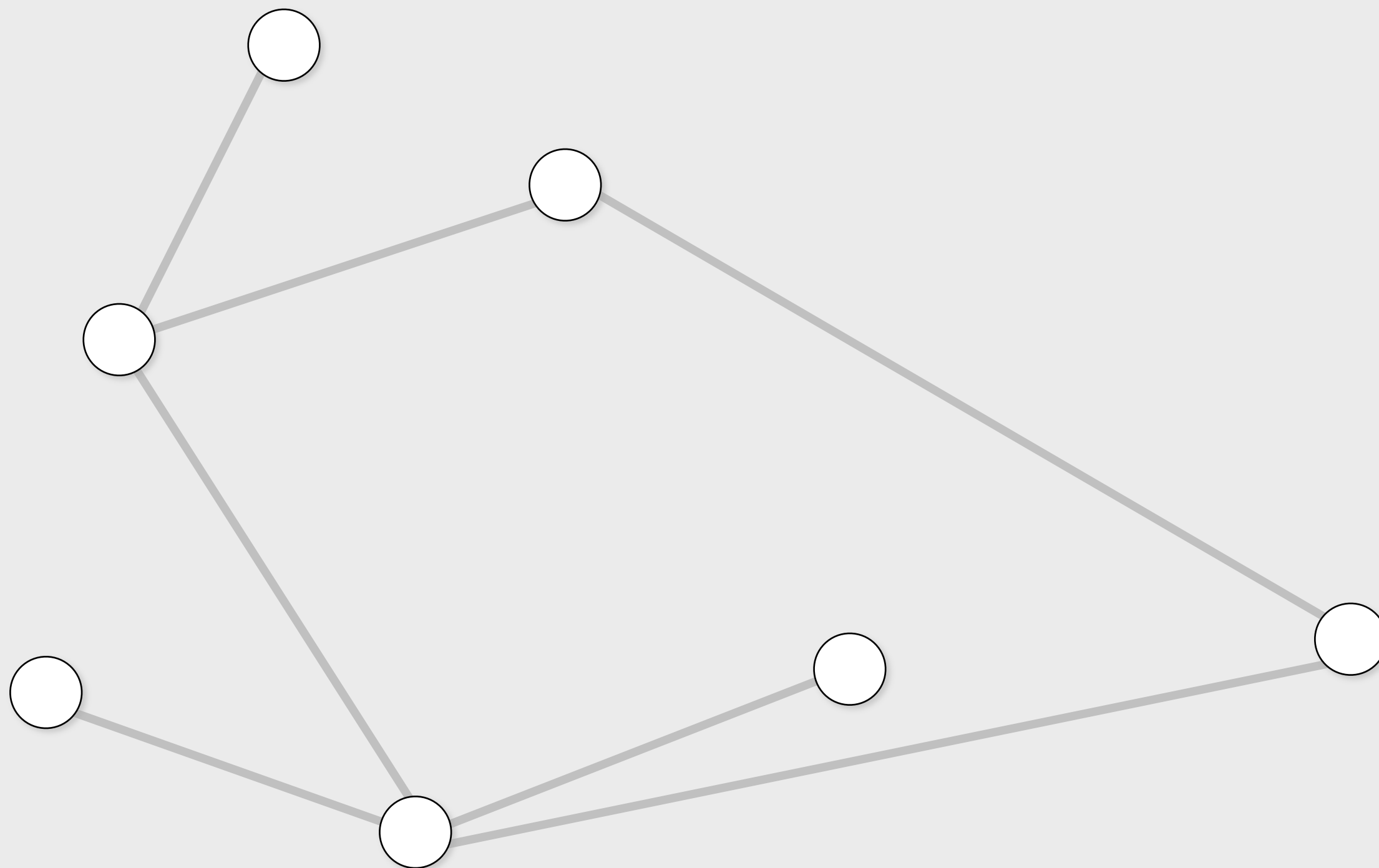


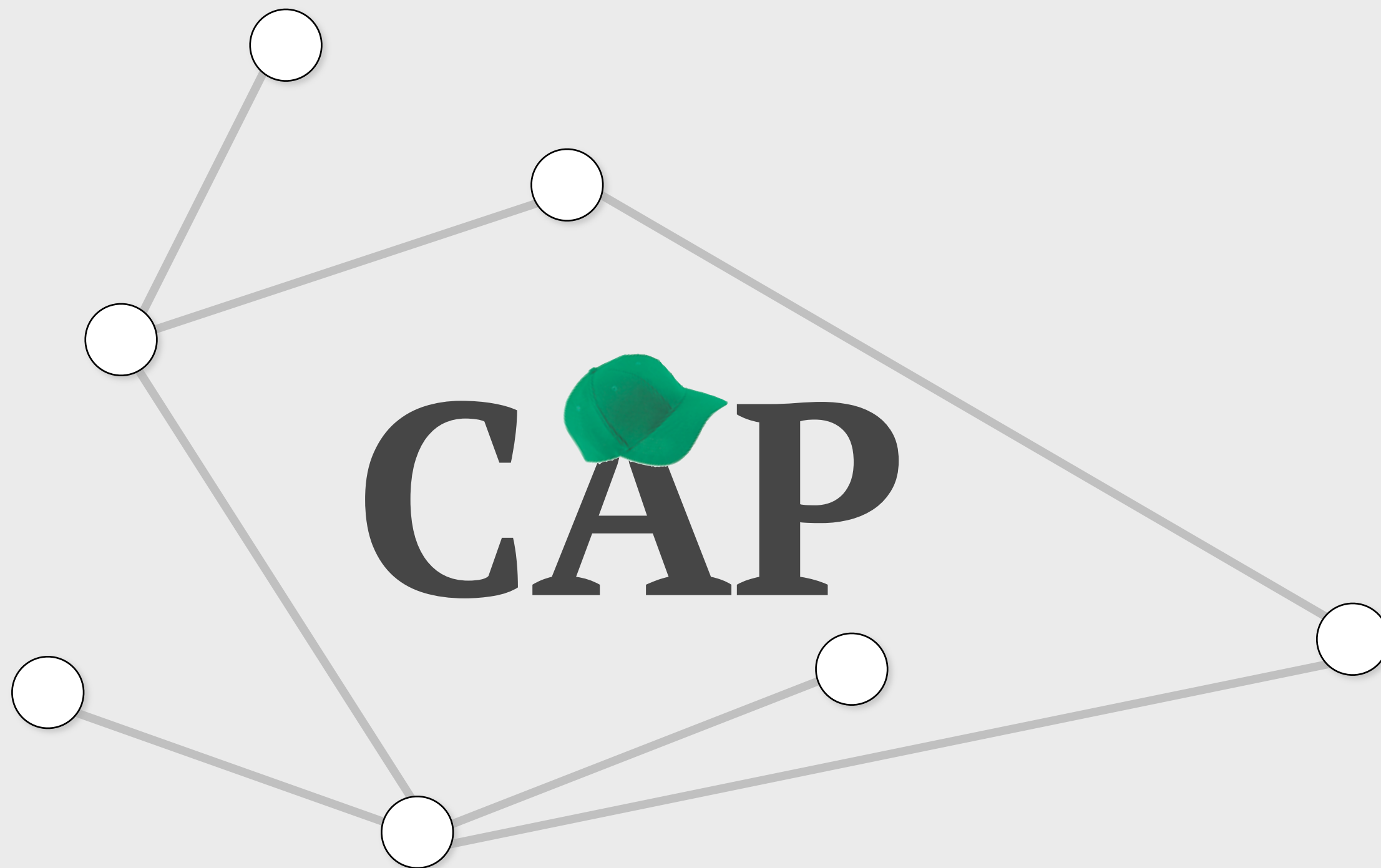
LVars:

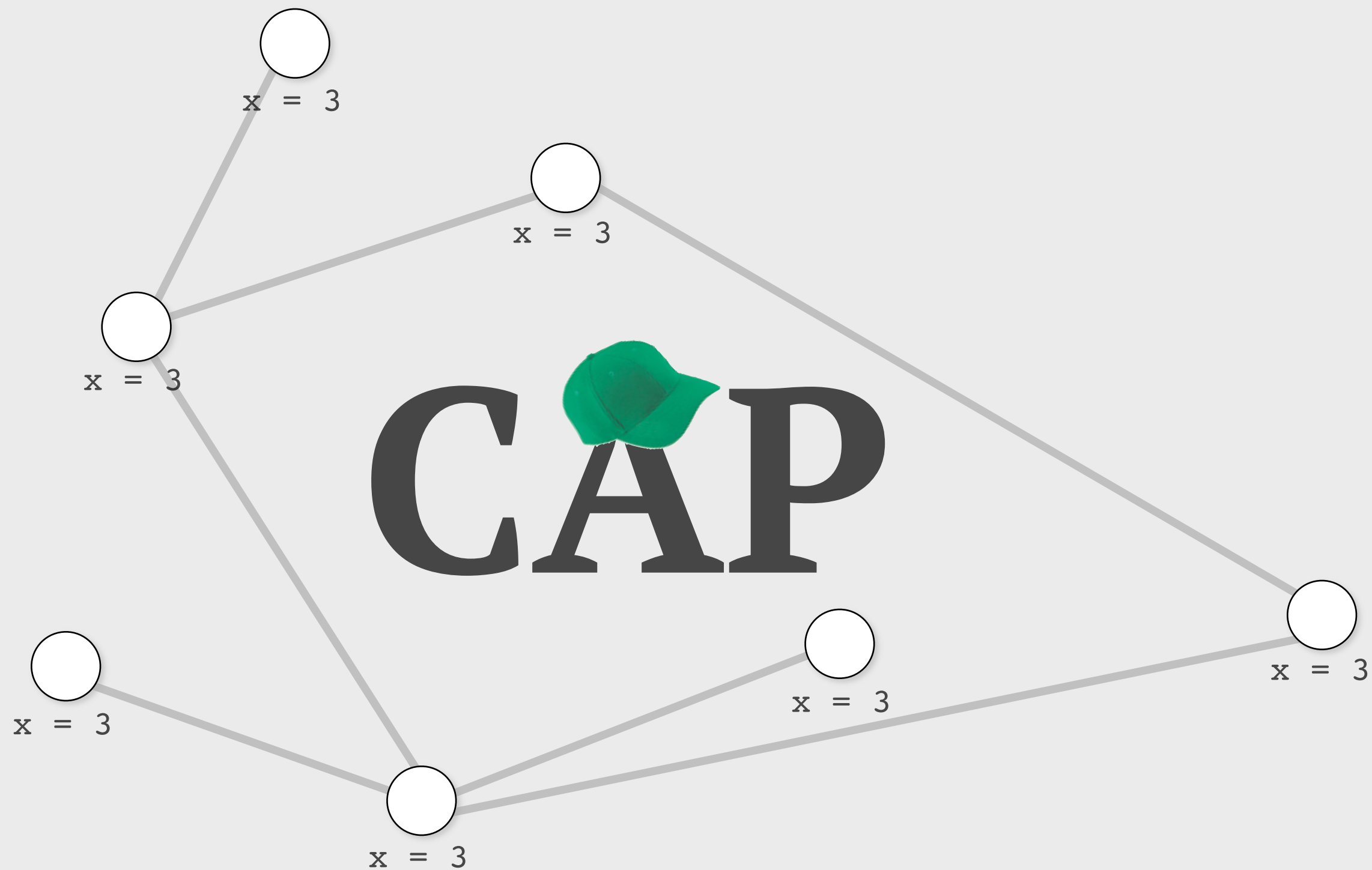
**Lattice-based Data Structures
for Deterministic Parallelism**

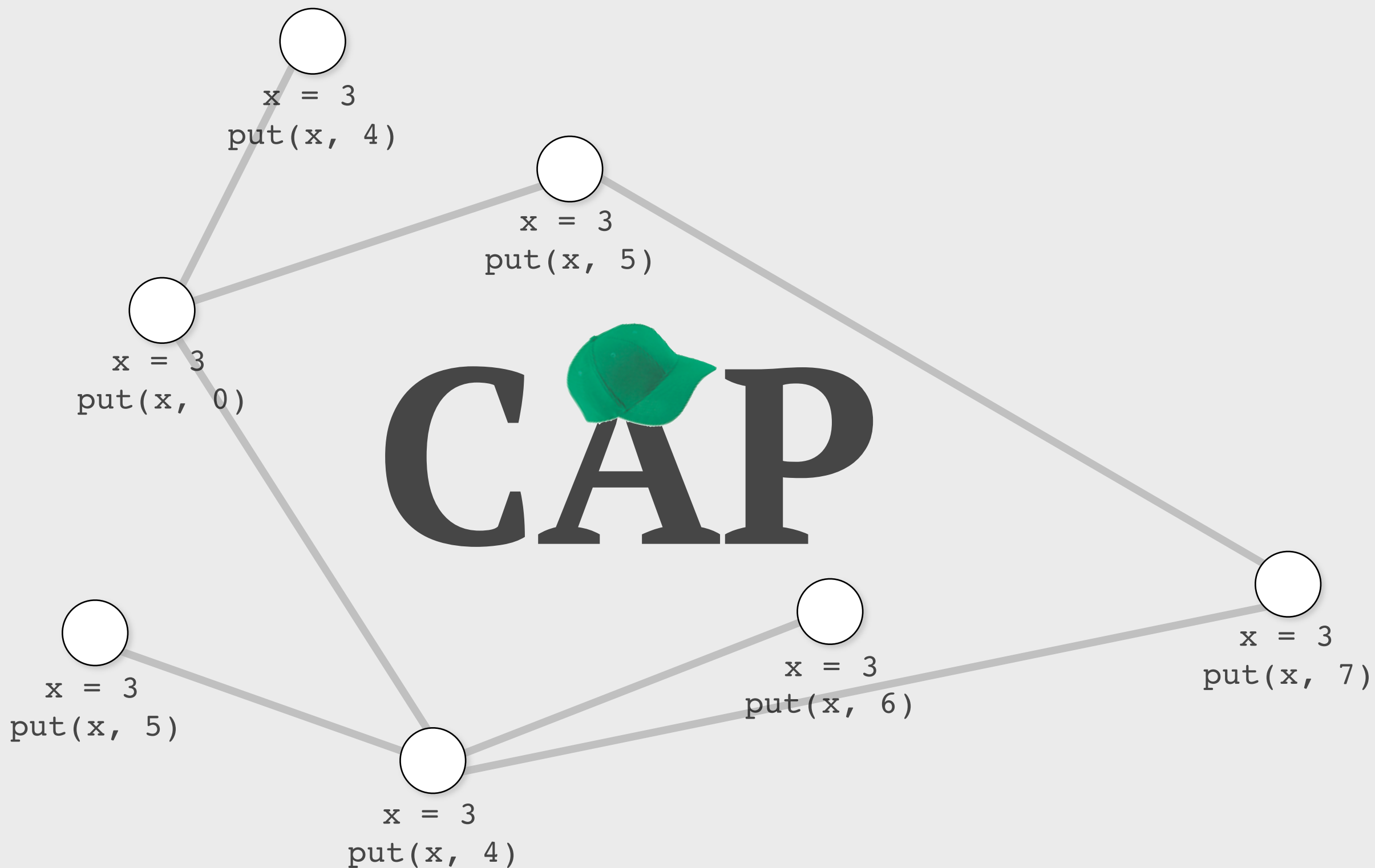
Lindsey Kuper
Indiana University

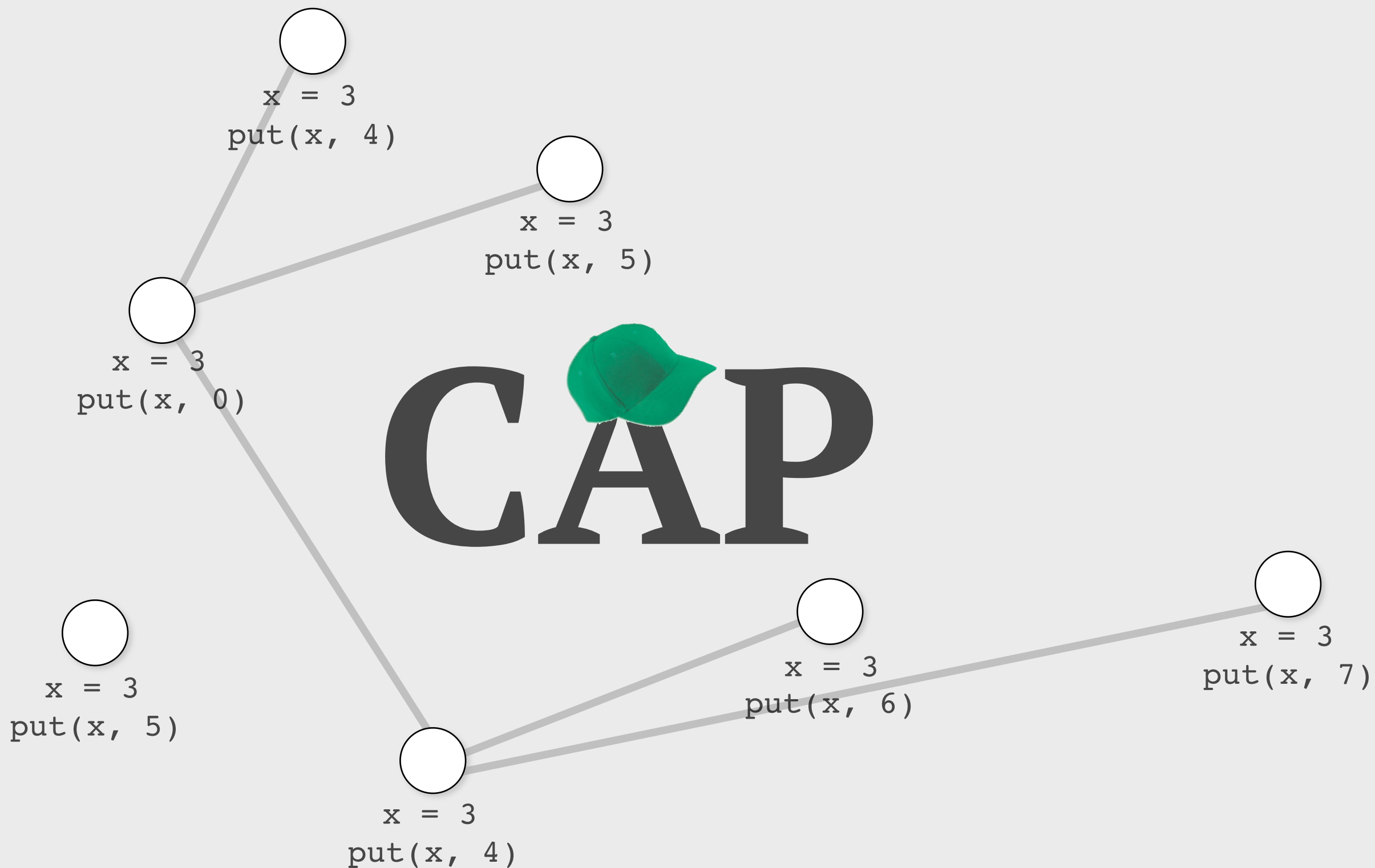
RICON West, San Francisco, CA, USA
October 29, 2013

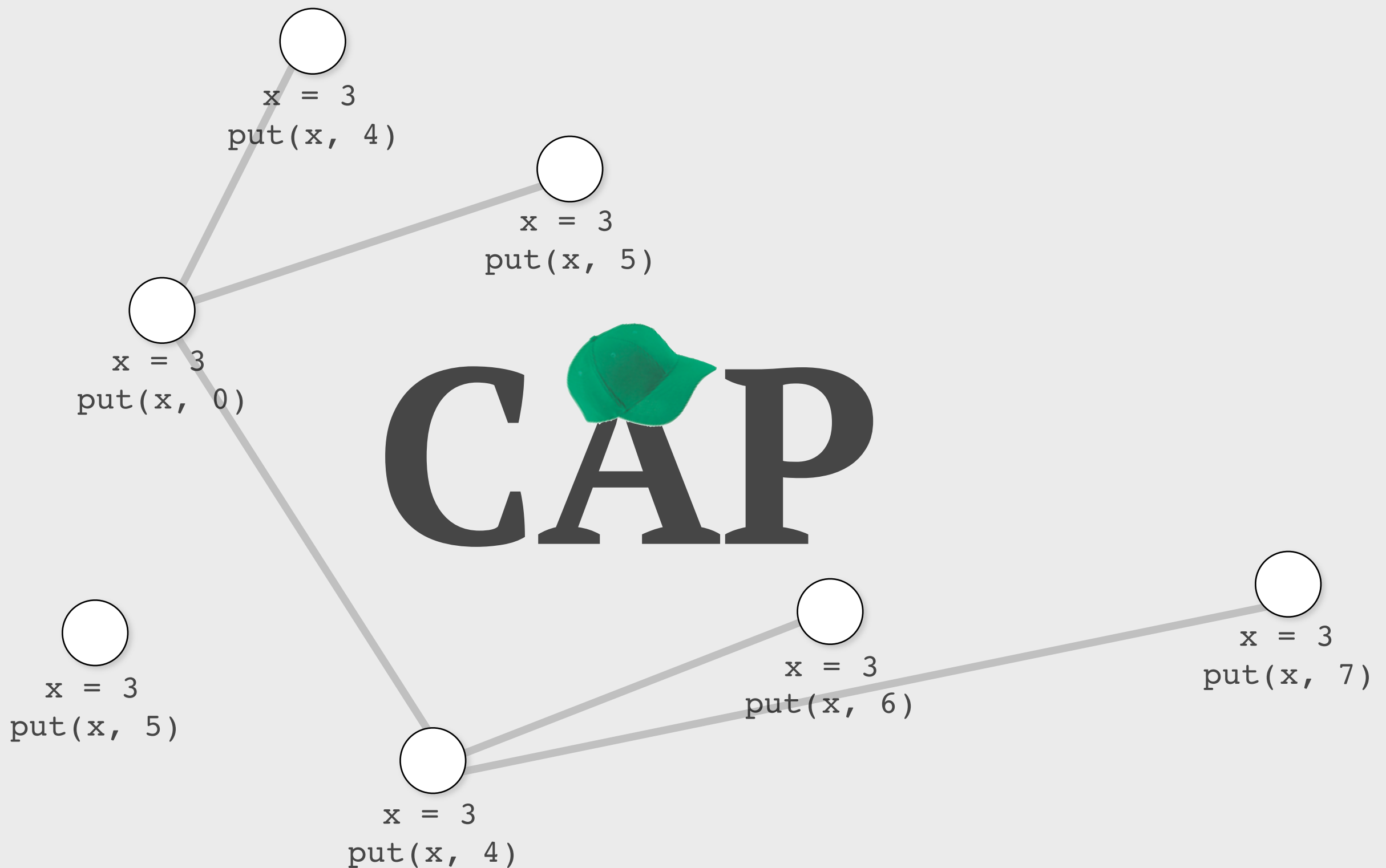












Eventual consistency.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management; D.4.5 [Operating Systems]: Reliability; D.4.2 [Operating Systems]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Eventual consistency.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able

add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being upgraded. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from a data store, and that its data needs to be available in multiple data centers.

Amazon's infrastructure is comprised of millions of servers in a standard mode of operation; there are always a significant number of server and network components that are down at any given time. As such Amazon's software must be constructed in a manner that treats failure as a normal case without impacting availability or

performance. Amazon has developed a family of storage technologies, of which the Amazon Simple Storage Service (S3) is also available outside of Amazon and known as probably the best known. This paper presents the implementation of Dynamo, another highly available distributed data store built for Amazon's platform. Dynamo manages the state of services that have very different requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and

performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to “merge” the conflicting versions and return a single unified shopping cart.

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.

Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro^{1,5}, Nuno Preguiça^{1,2}, Carlos Baquero³, and Marek Zawirski^{1,4}

¹ INRIA, Paris, France

² CITI, Universidade Nova de Lisboa, Portugal

³ Universidade do Minho, Portugal

⁴ UPMC, Paris, France

⁵ LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able

add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being upgraded. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from the data store, and that its data needs to be available across multiple data centers.

Amazon’s infrastructure is comprised of millions of servers in a standard mode of operation; there are always a significant number of server and network components that fail at any given time. As such Amazon’s software must be constructed in a manner that treats failure as a normal case without impacting availability or

ware of the data schema it uses. The method that is best suited for this application that maintains consistency is to “merge” the conflicting updates into the shopping cart.

ability and scaling needs, Amazon has developed a set of storage technologies, of which the Amazon Simple Storage Service (S3) is also available outside of Amazon and known as probably the best known. This paper presents the implementation of Dynamo, another highly available distributed data store built for Amazon’s platform. Dynamo manages the state of services that have very different requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and

performance. Amazon’s platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon’s platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions of customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon’s platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP’07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.

Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro^{1,5}, Nuno Preguiça^{1,2}, Carlos Baquero³, and Marek Zawirski^{1,4}

¹ INRIA, Paris, France

² CITI, Universidade Nova de Lisboa, Portugal

³ Universidade do Minho, Portugal

⁴ UPMC, Paris, France

⁵ LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world. The consequences of a platform failure are immense.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed.

Amazon.com is a loosely coupled, service-oriented architecture consisting of hundreds of services. In this paper, we describe the design and implementation of storage technologies used by Amazon.com. Customers should be able to use the service even if disks are failing or data centers are being upgraded. The service is responsible for ensuring that it can always write to and read from the data needs to be available.

Amazon.com is comprised of millions of servers in operation; there are always a few server and network components that fail. Such as Amazon’s software infrastructure, a manner that treats failure as an expected event impacting availability or

Amazon.com has developed a key-value store which the Amazon Simple Storage Service (S3) and known as Dynamo. This paper presents the design and implementation of another highly available key-value store built for Amazon’s platform. The store is composed of services that have very different requirements: tight control over the system, high availability, cost-effectiveness and as a very diverse set of requirements. A select set of services that is flexible enough to support their data store appropriately and have high availability and cost-effective manner.

Amazon’s platform that only need to be replicated for many services, such as

those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP’07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.

Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro^{1,5}, Nuno Preguiça^{1,2}, Carlos Baquero³, and Marek Zawirski^{1,4}

¹ INRIA, Paris, France

² CITI, Universidade Nova de Lisboa, Portugal

³ Universidade do Minho, Portugal

⁴ UPMC, Paris, France

⁵ LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world. The consequences of a platform failure are immense.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed.

Amazon.com is a loosely coupled, service-oriented architecture consisting of hundreds of services. In this paper, we describe the design and implementation of storage technologies that enable customers to be able to add items to their shopping cart even if disks are failing or data centers are being upgraded. The service responsible for the cart must be able to always write to and read from the data needs to be available

Amazon.com is comprised of millions of servers in operation; there are always a few server and network components that fail. As such Amazon’s software must be designed in a manner that treats failure as a normal event, not impacting availability or

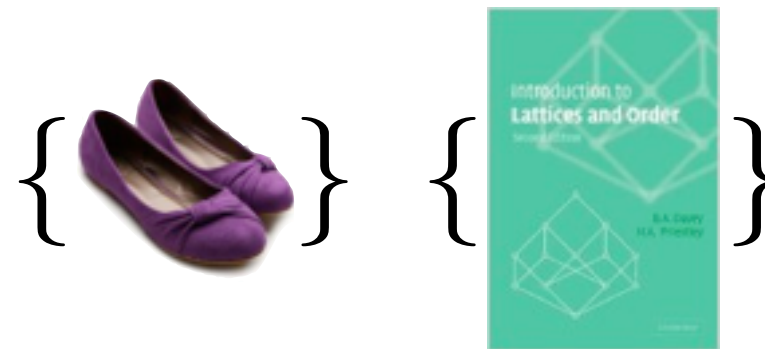
Amazon.com has developed a key-value store which the Amazon Simple Storage Service (S3) and known as Dynamo. This paper presents the design and implementation of another highly available key-value store built for Amazon’s platform. It is a set of services that have very different requirements: tight control over the system, high efficiency, cost-effectiveness and as a very diverse set of requirements. A select set of services that is flexible enough to handle their data store appropriately in a high availability and cost effective manner.

Amazon’s platform that only need to be replicated for many services, such as

those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP’07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.



Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro^{1,5}, Nuno Preguiça^{1,2}, Carlos Baquero³, and Marek Zawirski^{1,4}

¹ INRIA, Paris, France

² CITI, Universidade Nova de Lisboa, Portugal

³ Universidade do Minho, Portugal

⁴ UPMC, Paris, France

⁵ LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world. The consequences of a platform failure are immense.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed.

Amazon.com is a loosely coupled, service-oriented architecture consisting of hundreds of services. In this paper, we describe the design and implementation of storage technologies used by Amazon.com. Customers should be able to use the service even if disks are failing or data centers are being upgraded. The service is responsible for ensuring that it can always write to and read from the data needs to be available.

Amazon.com is comprised of millions of servers in operation; there are always a number of server and network components that fail. Such as Amazon’s software infrastructure, a failure in a manner that treats failure as impacting availability or

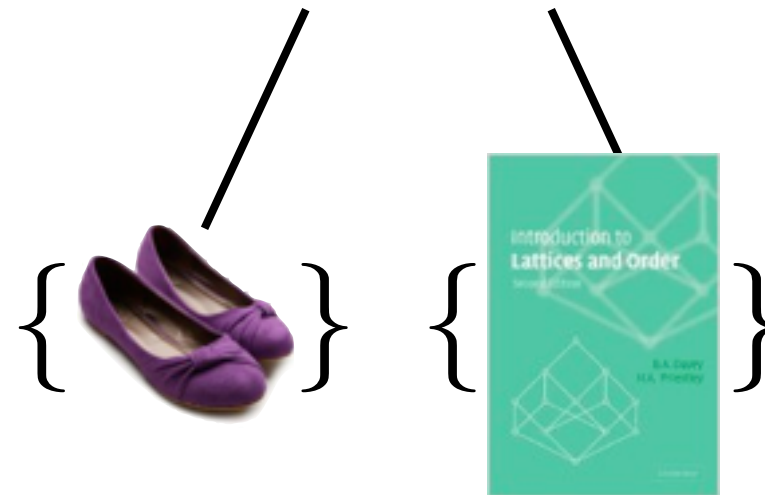
Amazon.com has developed a system in which the Amazon Simple Storage Service (S3) is the backbone of Amazon and known as Amazon S3. This paper presents the design and implementation of another highly available and scalable key-value store for Amazon’s platform. The store is composed of services that have very different requirements. Amazon has tight control over the system’s efficiency, cost-effectiveness and as a very diverse set of requirements. A select set of technologies that is flexible enough to handle their data store appropriately in a high availability and cost-effective manner.

Amazon’s platform that only need to be replicated for many services, such as

those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP’07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.



Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro^{1,5}, Nuno Preguiça^{1,2}, Carlos Baquero³, and Marek Zawirski^{1,4}

¹ INRIA, Paris, France

² CITI, Universidade Nova de Lisboa, Portugal

³ Universidade do Minho, Portugal

⁴ UPMC, Paris, France

⁵ LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world. The consequences of a platform failure are immense.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed.

Amazon.com is a loosely coupled, service-oriented architecture comprised of hundreds of services. In this paper, we describe the design and implementation of storage technologies used by Amazon.com. Customers should be able to use our services even if disks are failing or data centers are being upgraded. The service responsible for storing a customer’s shopping cart must always be available and able to write to and read from the data needed to be available.

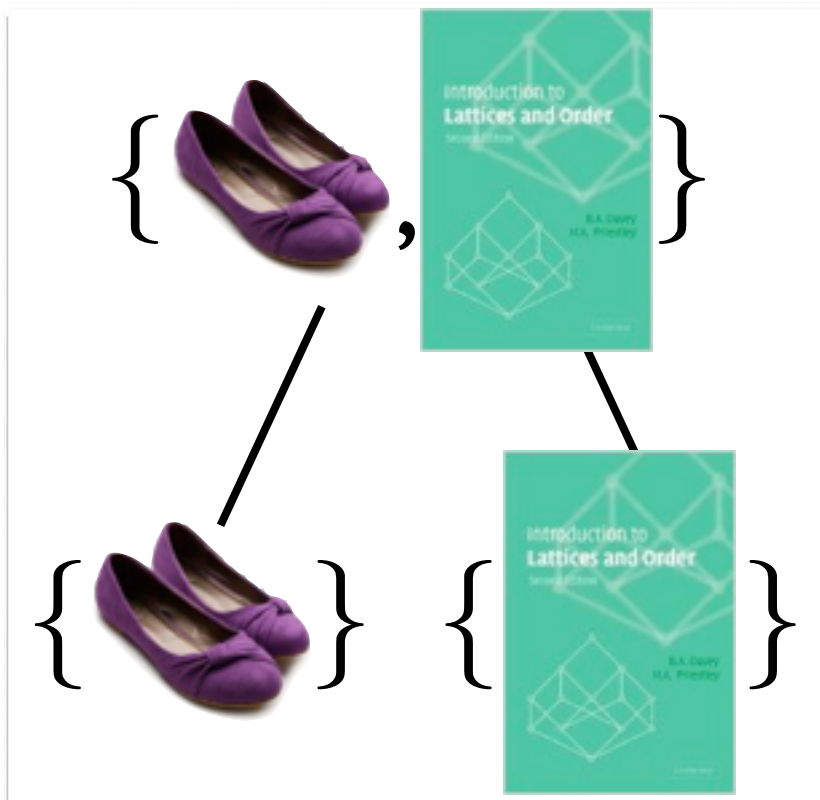
Amazon.com is comprised of millions of servers in operation; there are always a number of server and network components that fail. As such, Amazon’s software must be designed in a manner that treats failure as a normal event, impacting availability or performance.

Amazon.com has developed a key-value store which the Amazon Simple Storage Service (S3) is based on. This paper presents the design and implementation of another highly available key-value store built for Amazon’s platform. It is a set of services that have very different requirements: tight control over the service’s latency, cost-effectiveness and scalability. As a very diverse set of services, Amazon.com requires a technology that is flexible enough to support a wide range of requirements. A select set of services requires a data store appropriately designed to provide high availability and performance in an effective manner.

Amazon.com’s platform that only need to be replicated for many services, such as

those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs



Amazon.com consists of millions of servers. The strict consistency model required for many applications would prevent the growth of the system. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...\$5.00.

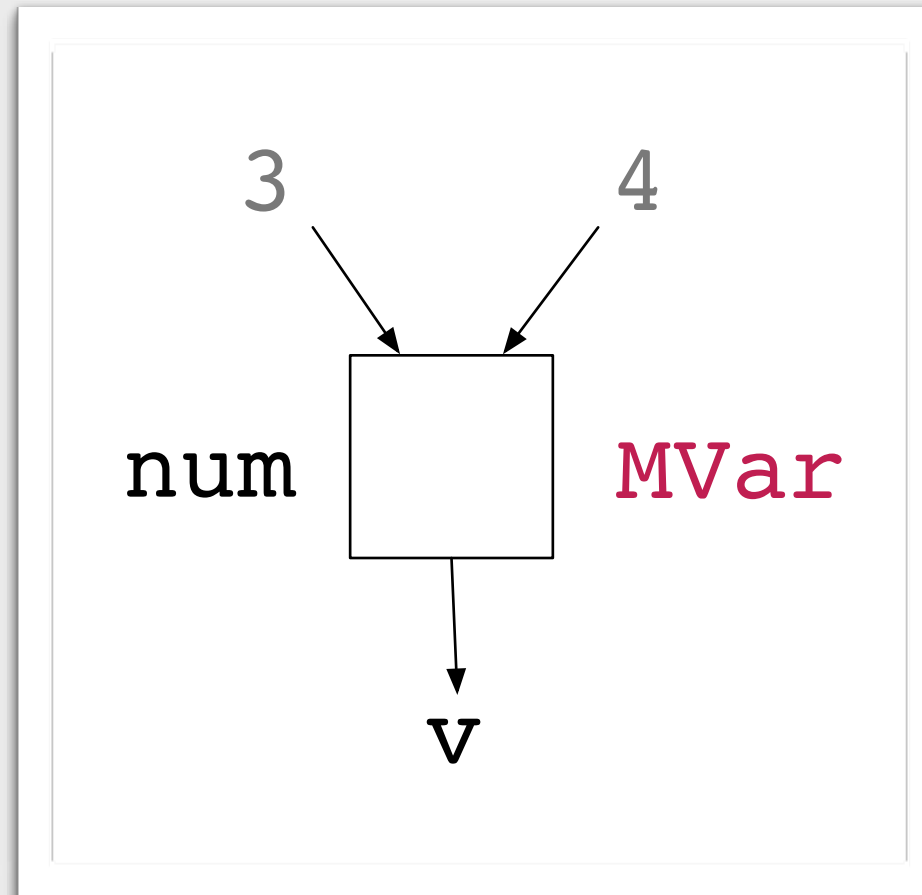
Eventual consistency.

Deterministic Parallelism

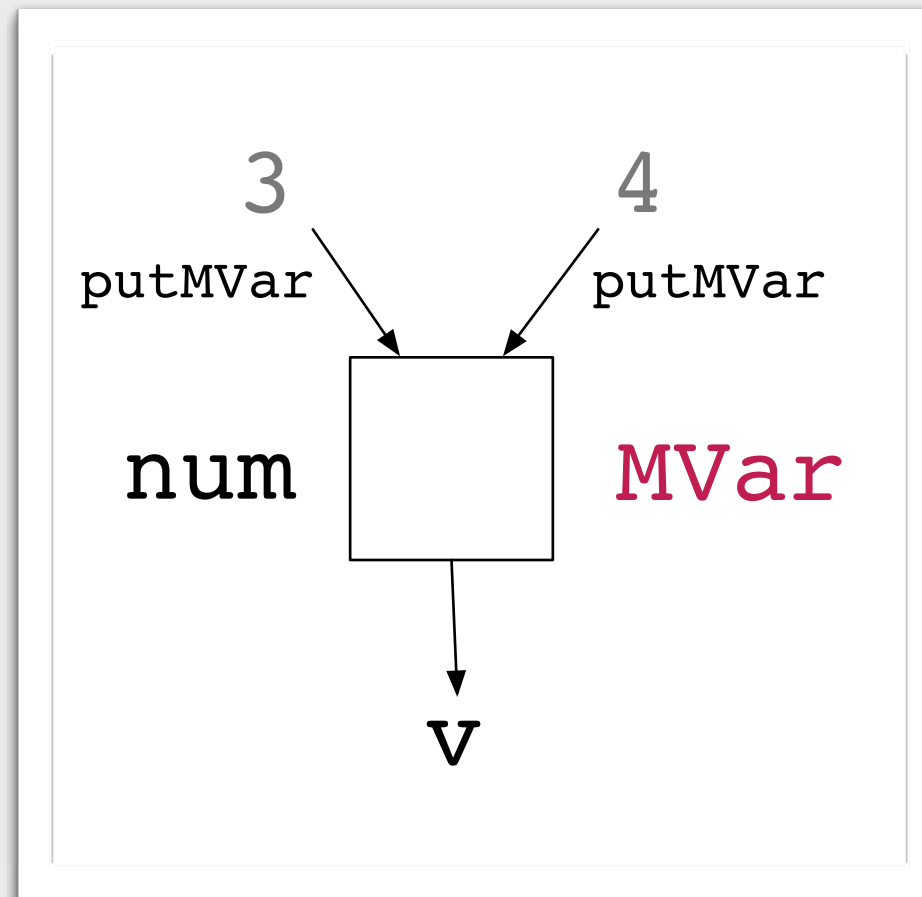
(observably)

Deterministic Parallelism

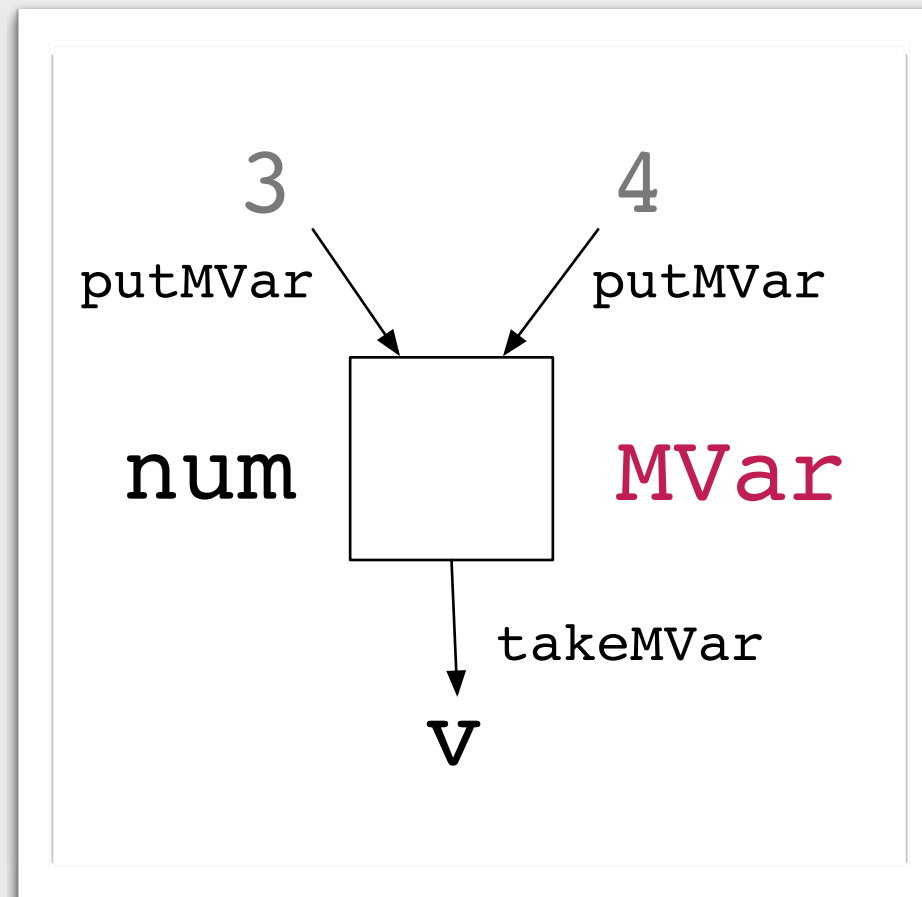
What does this program do?



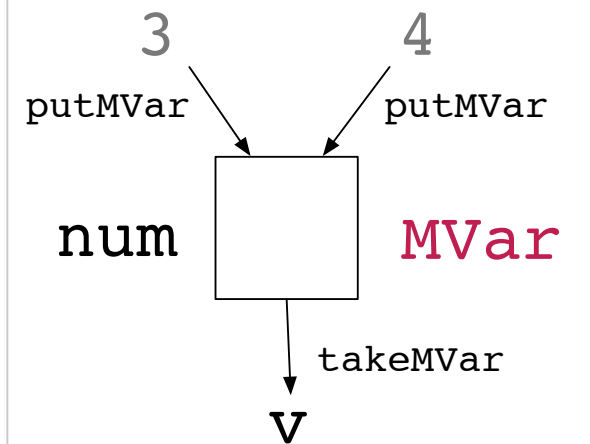
What does this program do?



What does this program do?

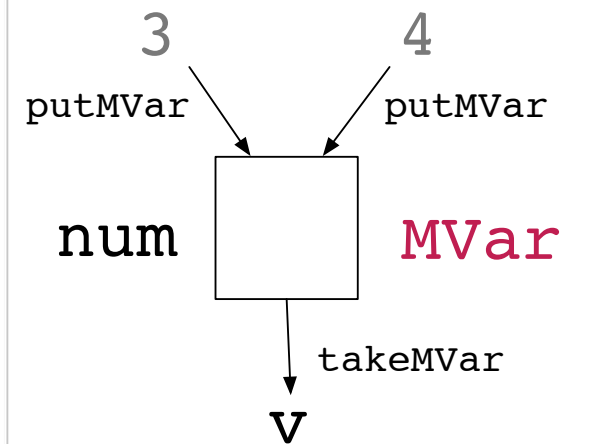


What does this program do?



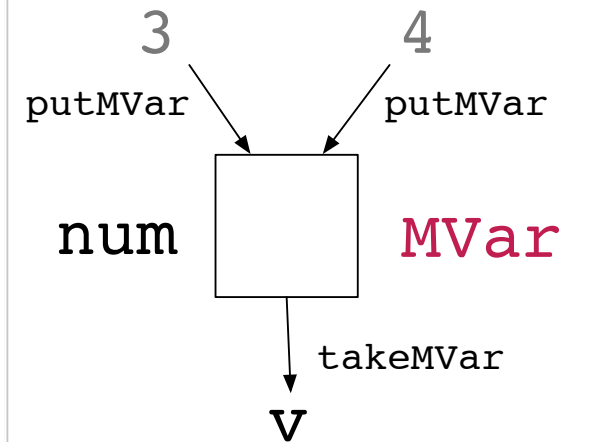
What does this program do?

`p = do`



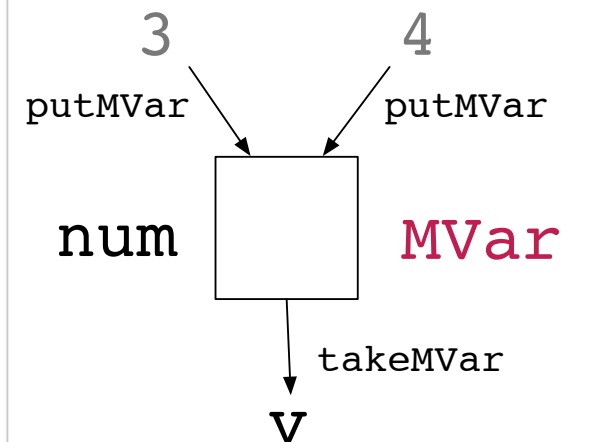
What does this program do?

```
p = do  
  num <- newEmptyMVar
```



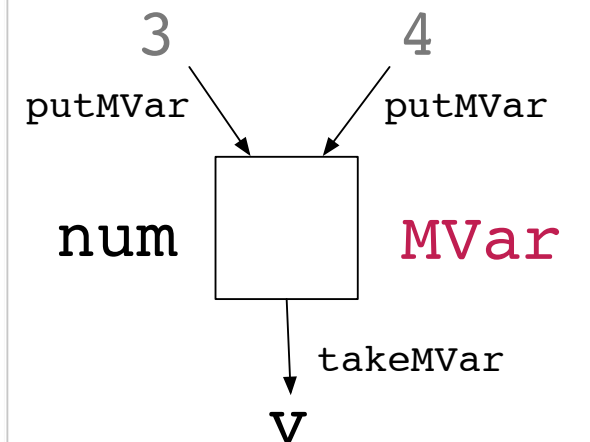
What does this program do?

```
p = do  
  num <- newEmptyMVar  
  forkIO (putMVar num 3)
```



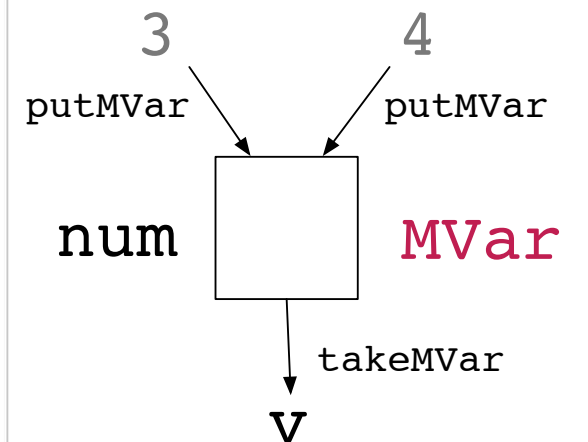
What does this program do?

```
p = do  
  num <- newEmptyMVar  
  forkIO (putMVar num 3)  
  forkIO (putMVar num 4)
```



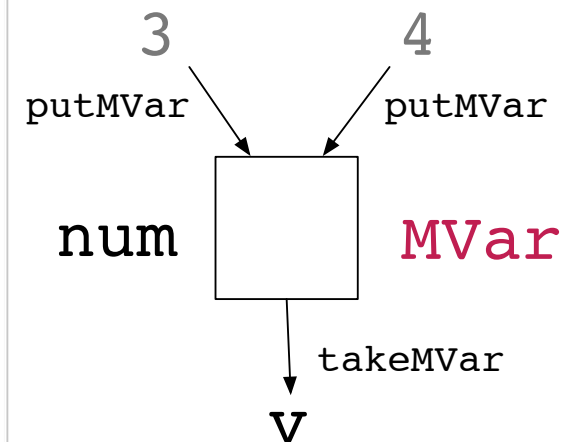
What does this program do?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
```



What does this program do?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```



[illegible]

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```


Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

```
p :: Par Int
p = do
  num <- new
  fork (put num 3)
  fork (put num 4)
  v <- get num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

```
p :: Par Int
p = do
  num <- new
  fork (put num 3)
  fork (put num 4)
  v <- get num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

```
./ivar-example +RTS -N2
ivar-example: multiple put
```

Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork (put num 3)
  fork (put num 4)
  v <- get num
  return v
```

Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork (put num 4)
  fork (put num 4)
  v <- get num
  return v
```


Deterministic programs that single-assignment forbids

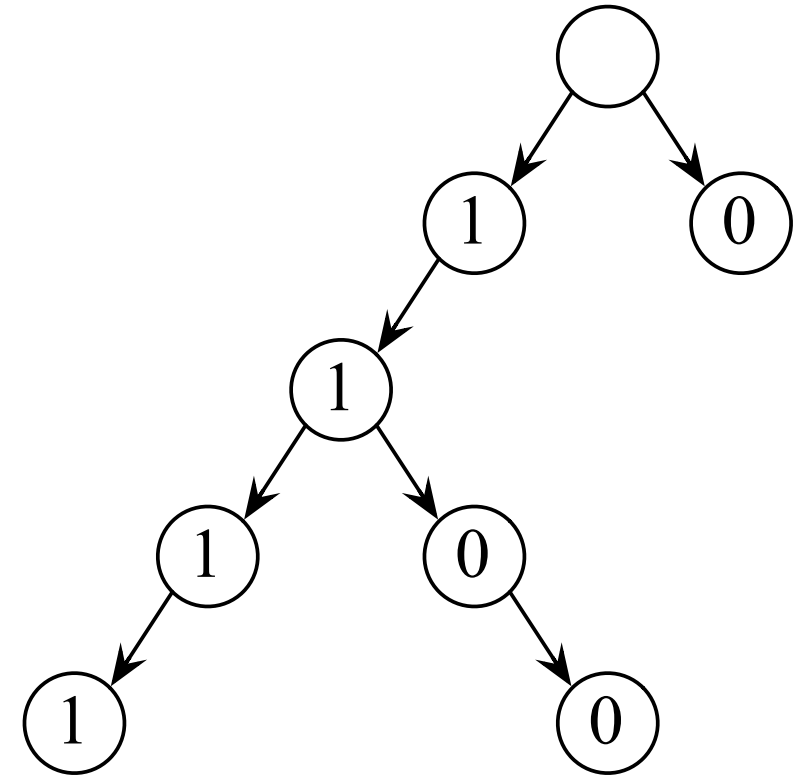
```
p :: Par Int
p = do
  num <- new
  fork (put num 4)
  fork (put num 4)
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork put num 4
  fork put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```



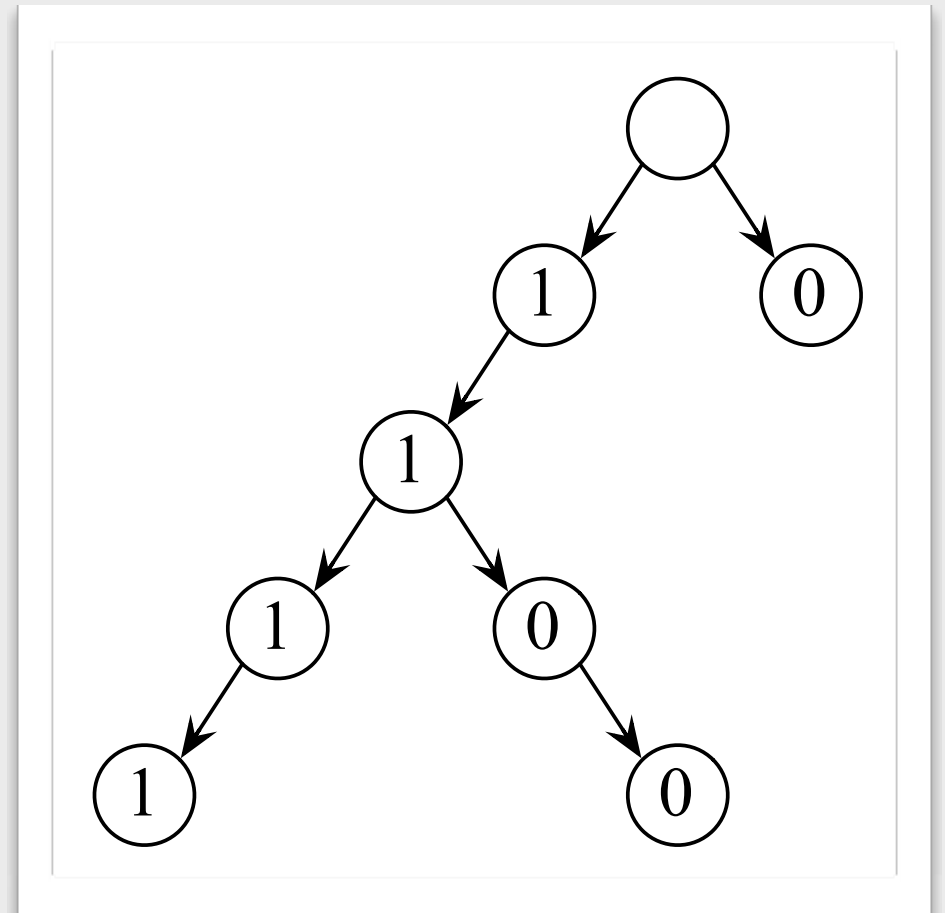
Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork (put num 4)
  fork (put num 4)
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

do

```
fork (insert t "0")
fork (insert t "1100")
fork (insert t "1111")
v <- get t
return v
```



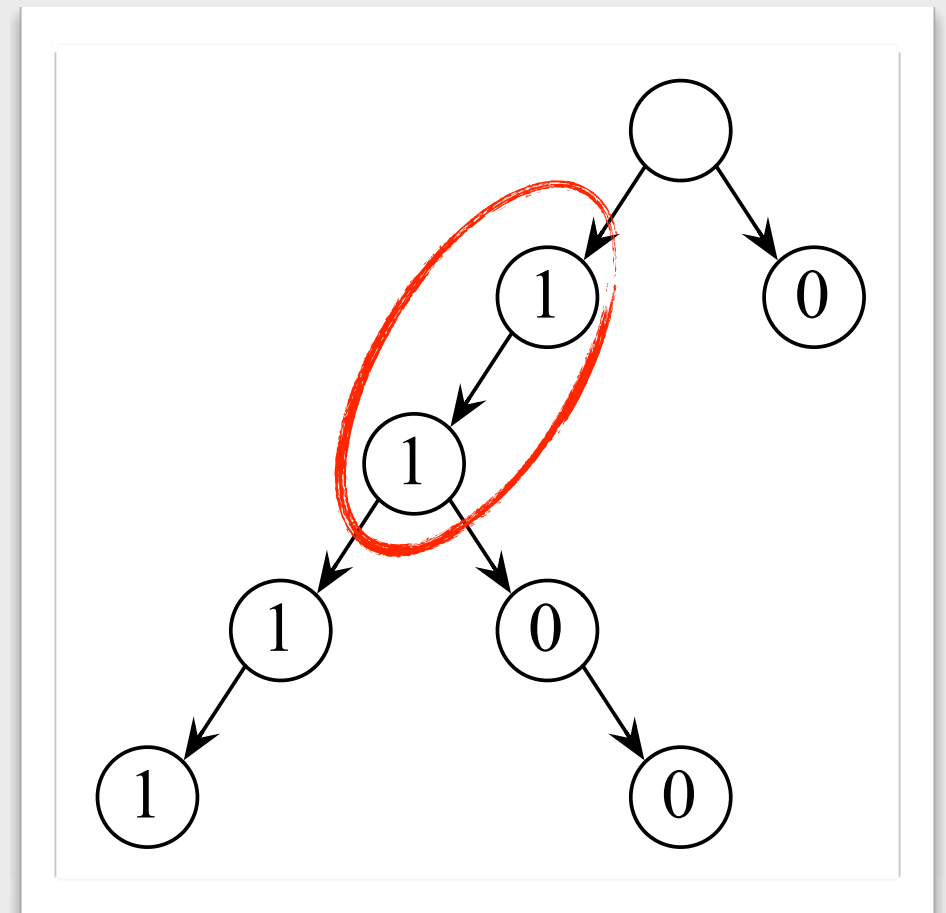
Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork (put num 4)
  fork (put num 4)
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

do

```
fork (insert t "0")
fork (insert t "1100")
fork (insert t "1111")
v <- get t
return v
```



LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]

LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write

LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```
import Control.LVish
import Data.LVar.Set
```

LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```
import Control.LVish  
import Data.LVar.Pair
```


LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```
import Control.LVish  
import Data.LVar.Map
```

LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```
import Control.LVish  
import Data.LVar.Counter
```

LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```
import Control.LVish
import Data.LVar.IVar
```

LVars: Multiple *monotonic* writes

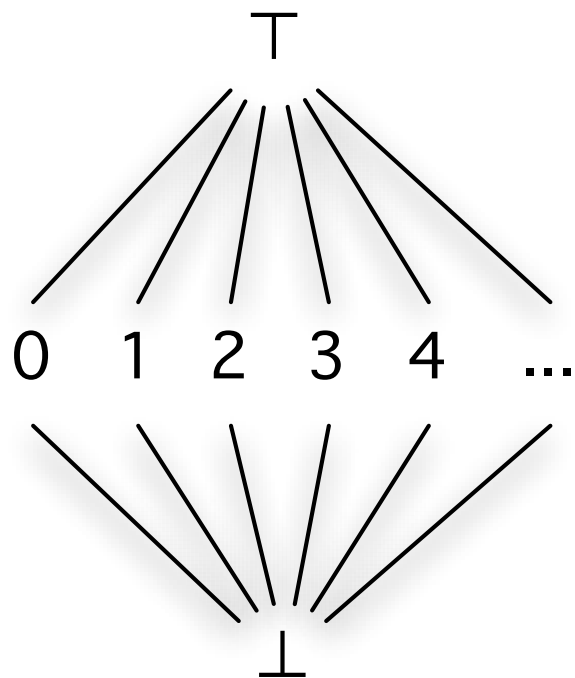
- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```
import Control.LVish
import Data.LVar.IVar
```

- `cabal install lvish today!`

LVars: Multiple *monotonic* writes

num



Raises an error, since $3 \sqcup 4 = \top$

do

```
fork (put num 3)
```

```
fork (put num 4)
```

Works fine, since $4 \sqcup 4 = 4$

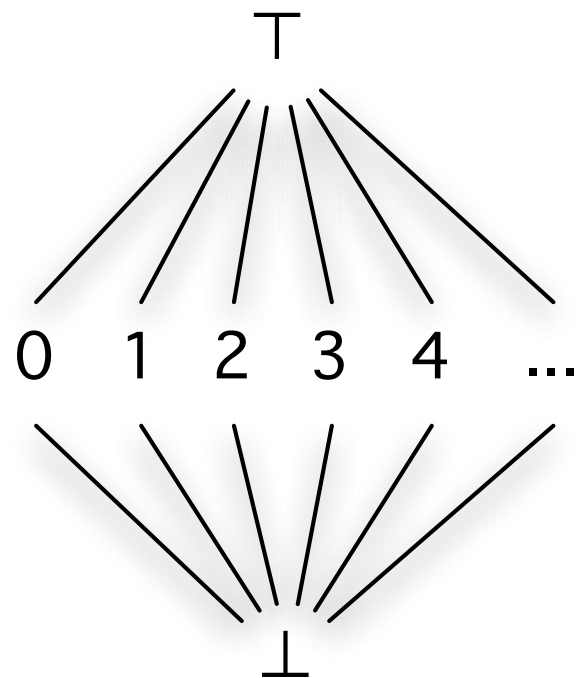
do

```
fork (put num 4)
```

```
fork (put num 4)
```

LVars: Multiple *monotonic* writes

num



Neil Conway
@neil_conway



Following

Immutability is a special case of monotone growth, albeit a particularly useful one.



Reply



Retweet



Favorited



More

7

RETWEETS

11

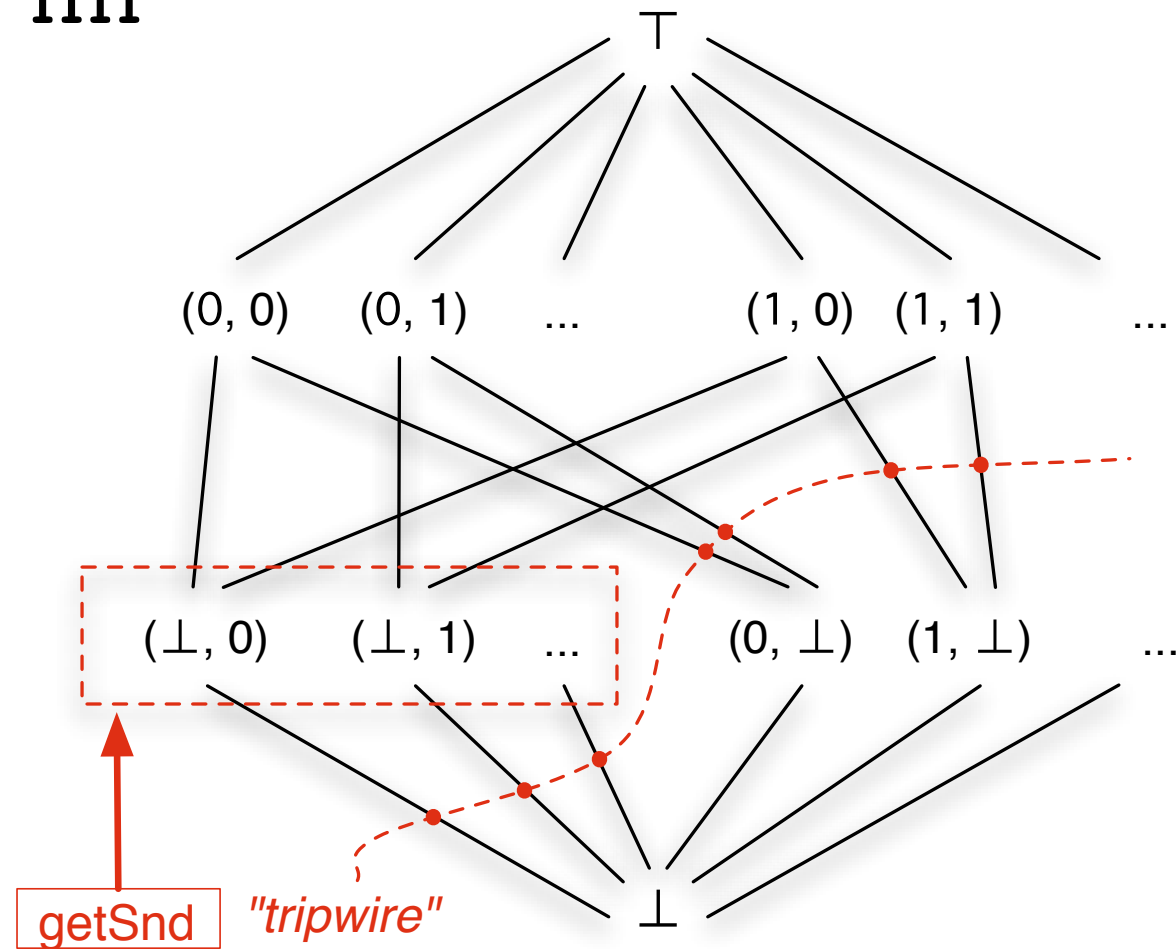
FAVORITES



10:10 AM - 21 Oct 13

LVars: Threshold reads

nn

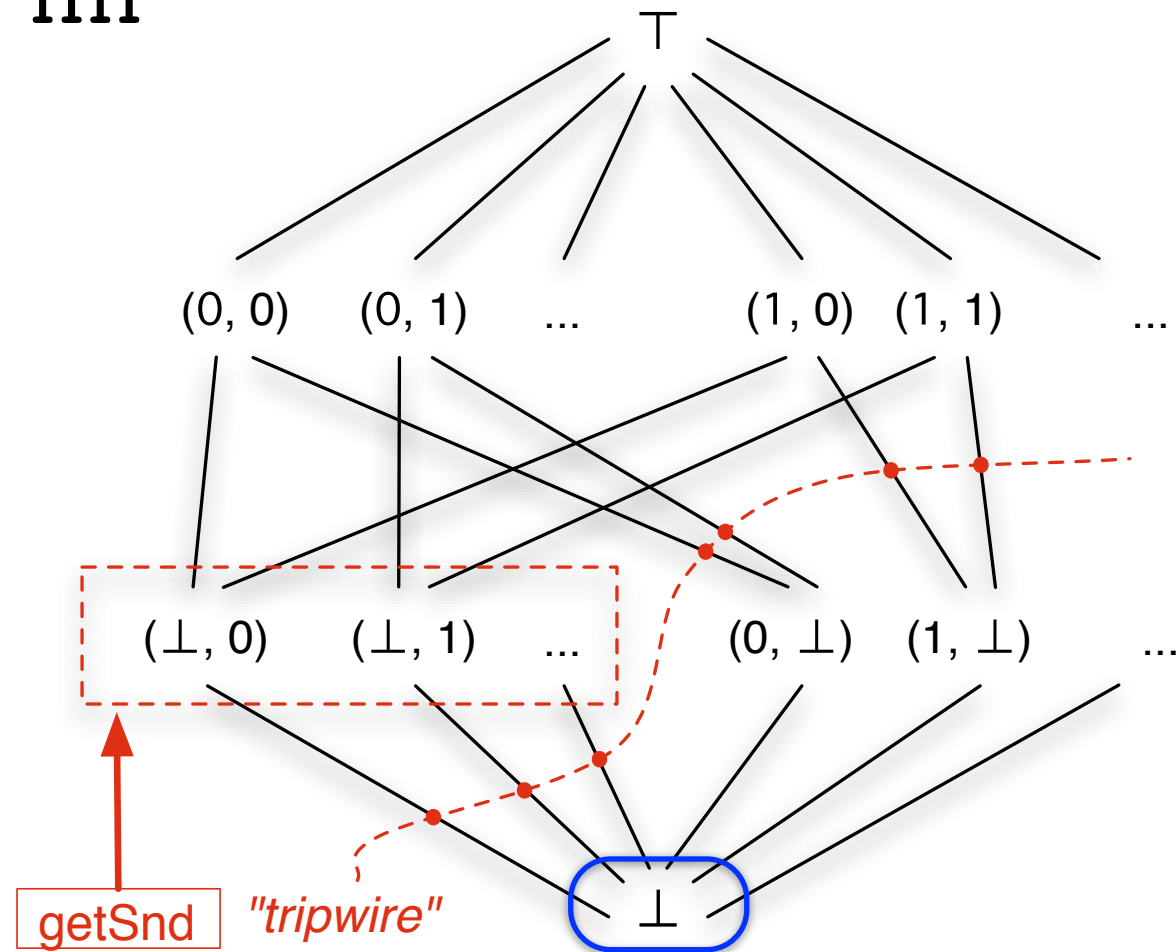


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```


LVars: Threshold reads

nn

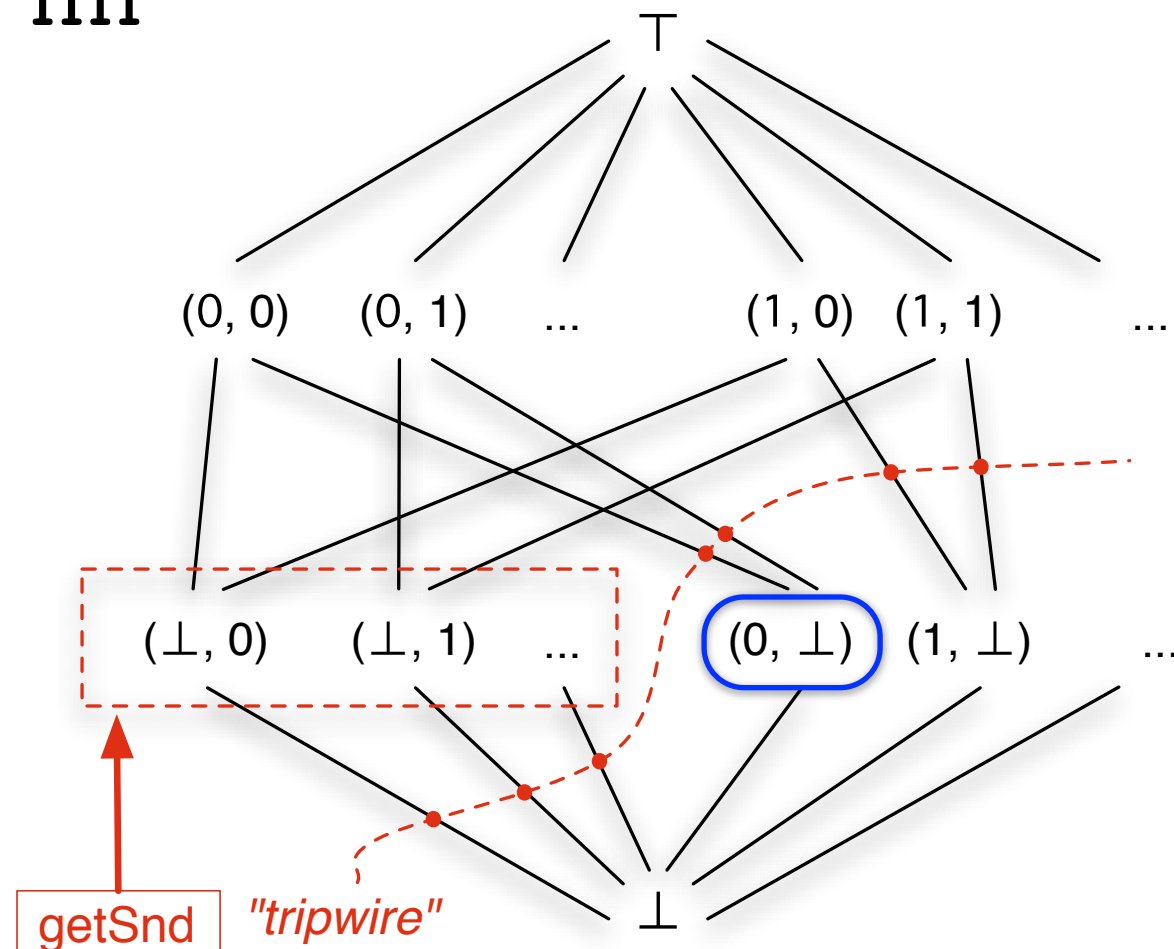


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

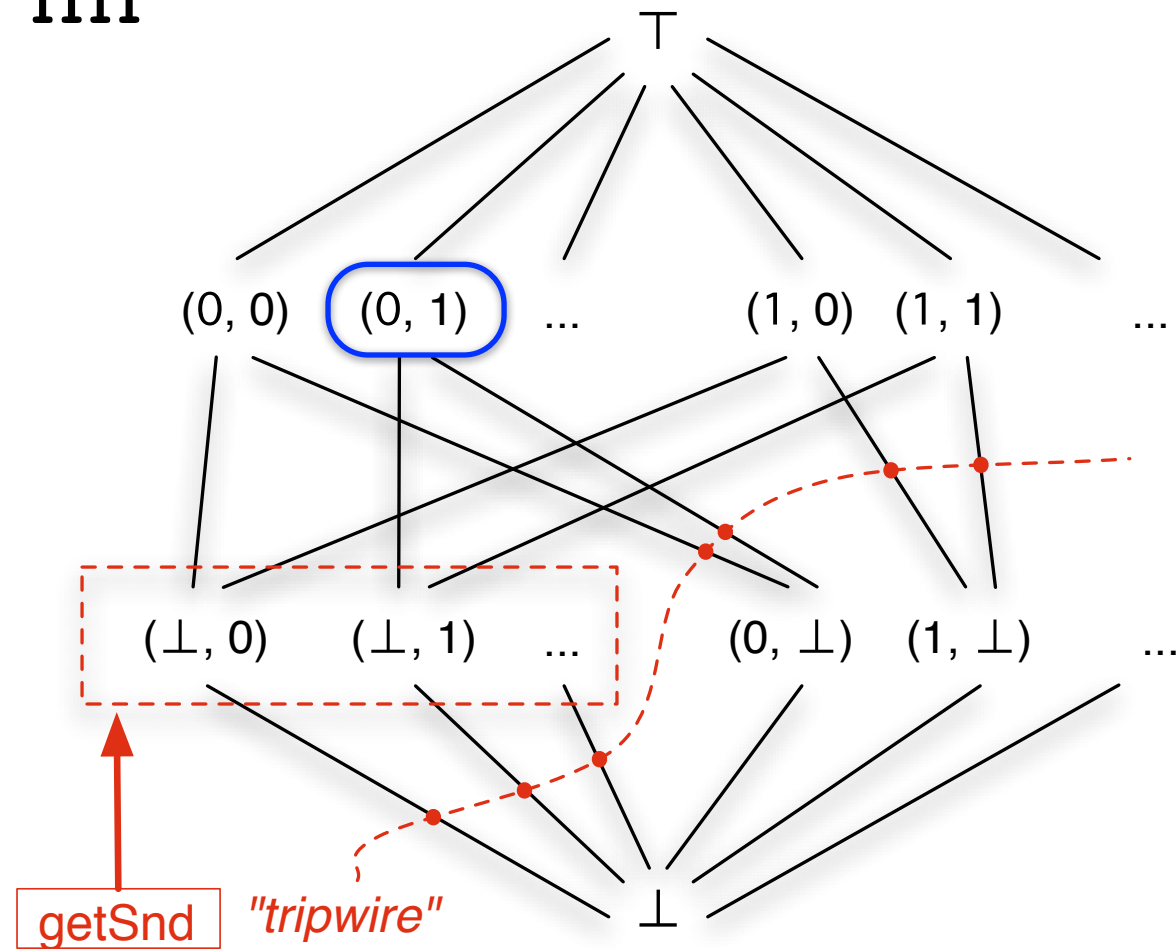


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

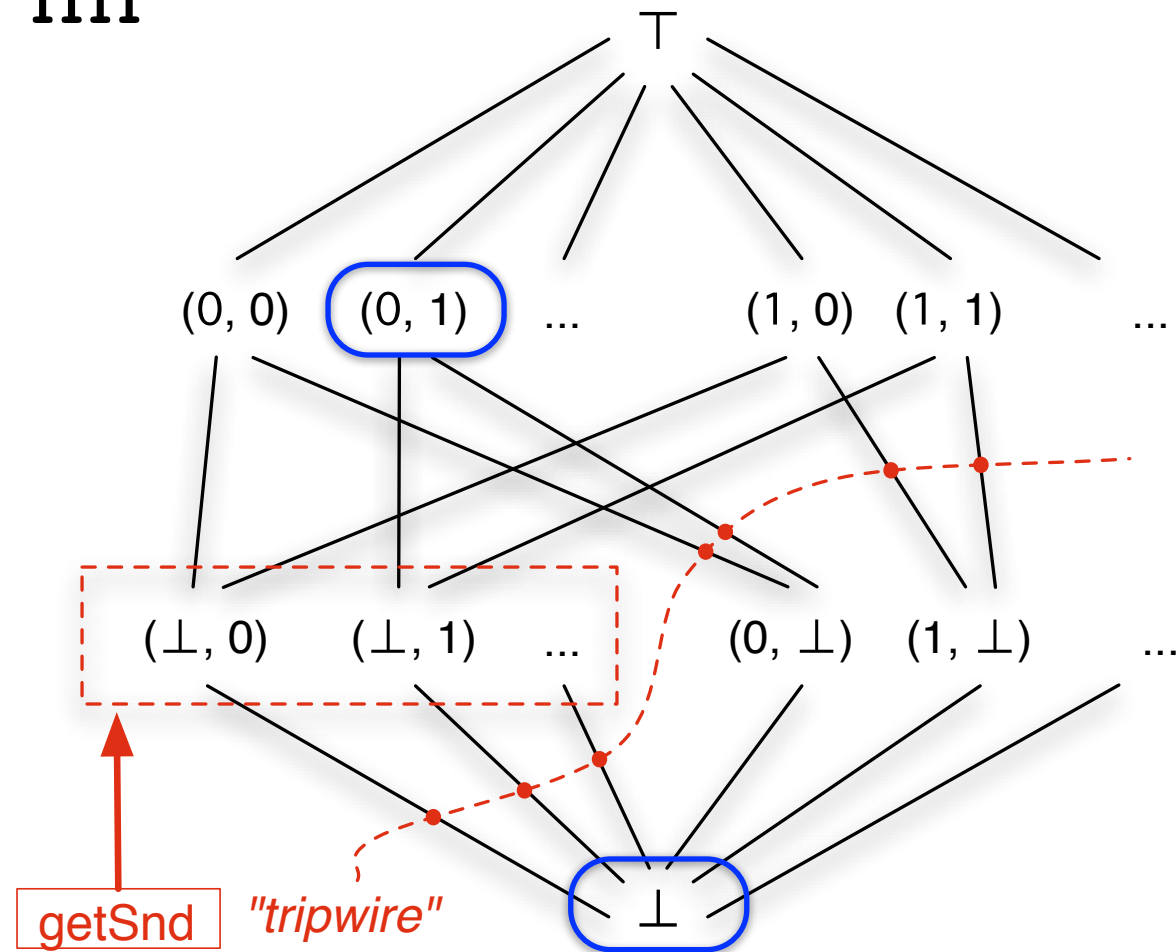


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

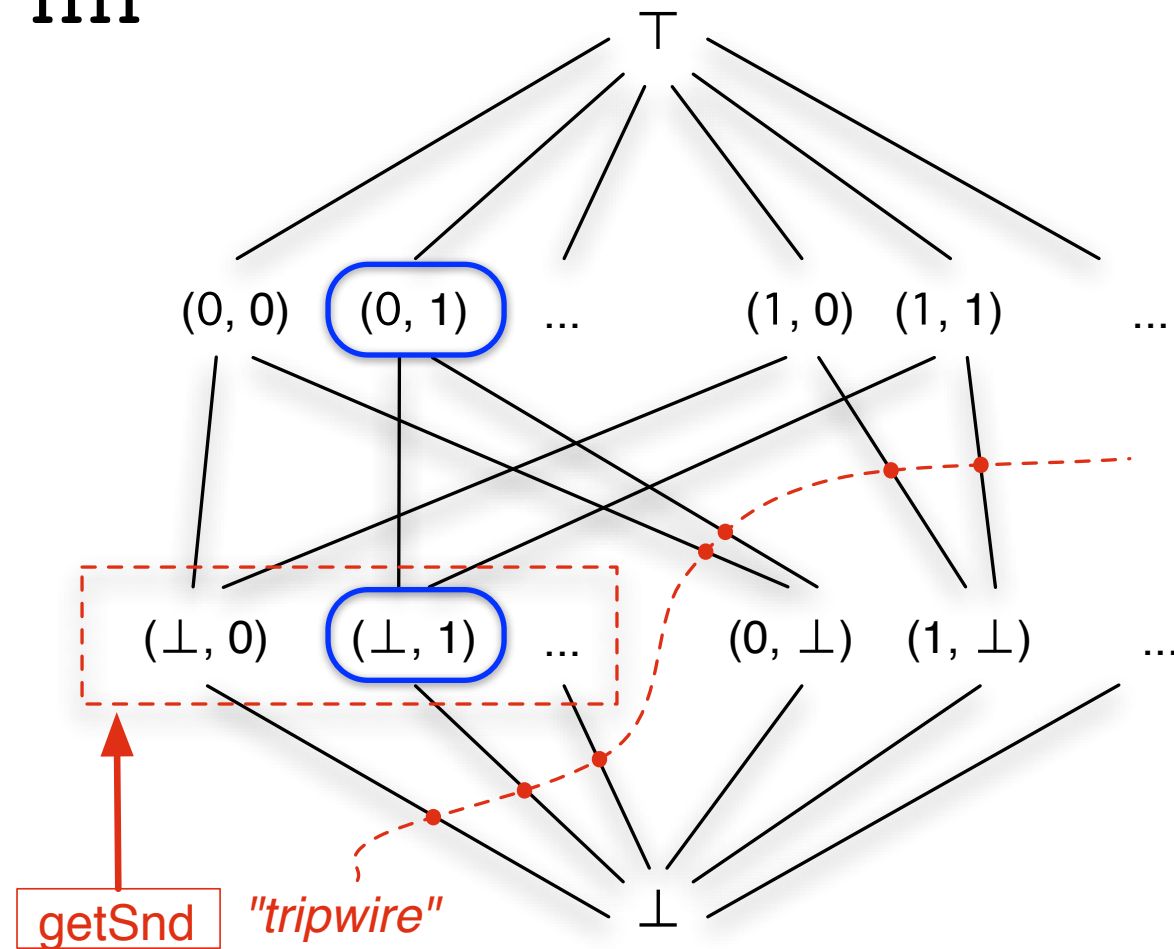


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

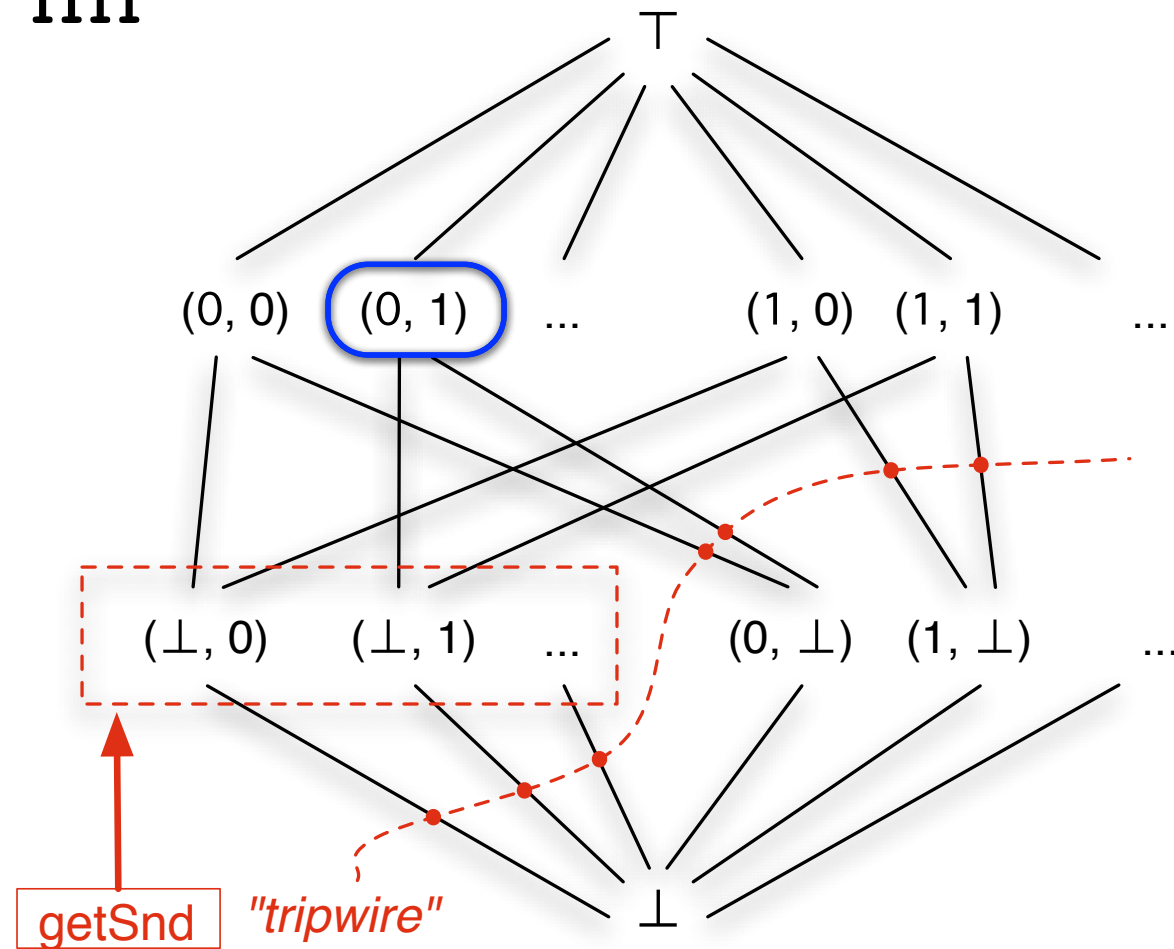


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```


LVars: Threshold reads

nn

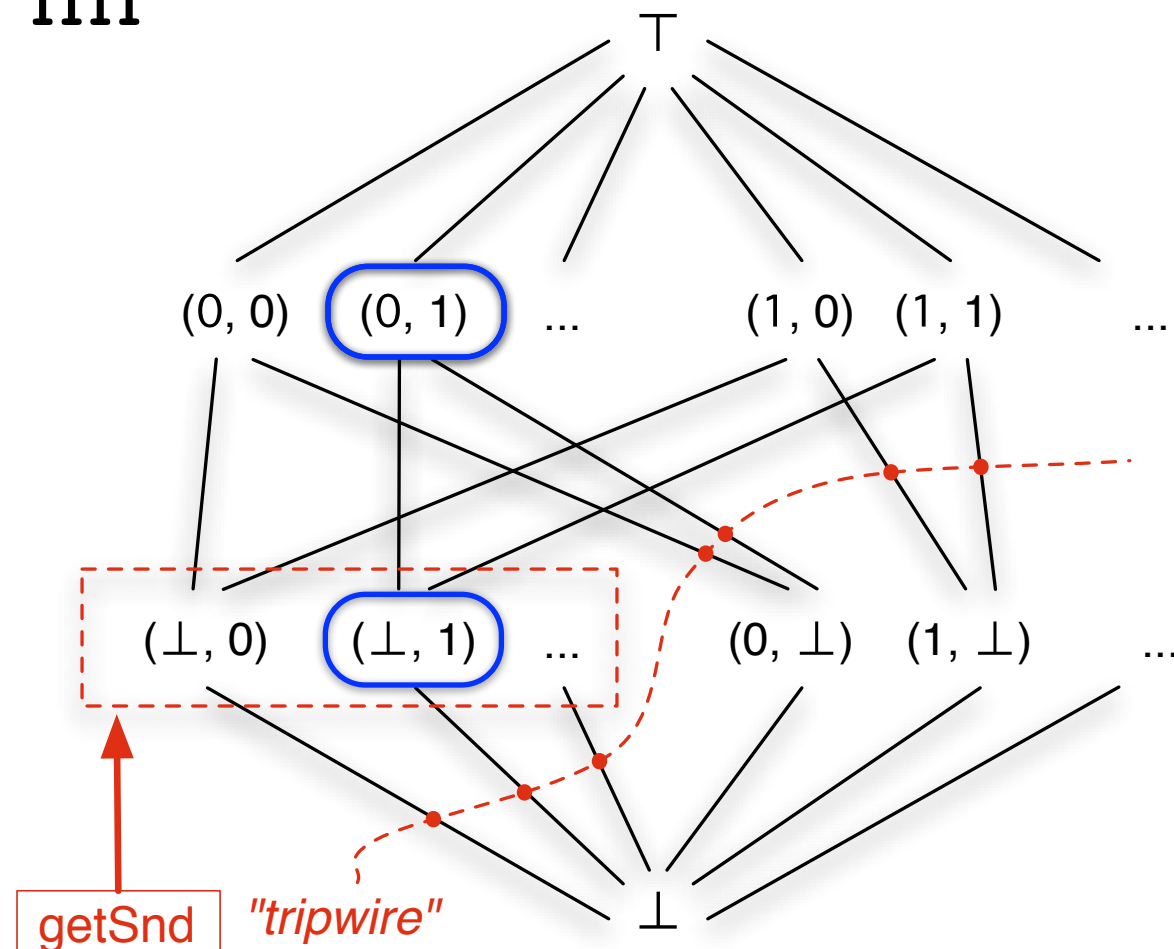


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn

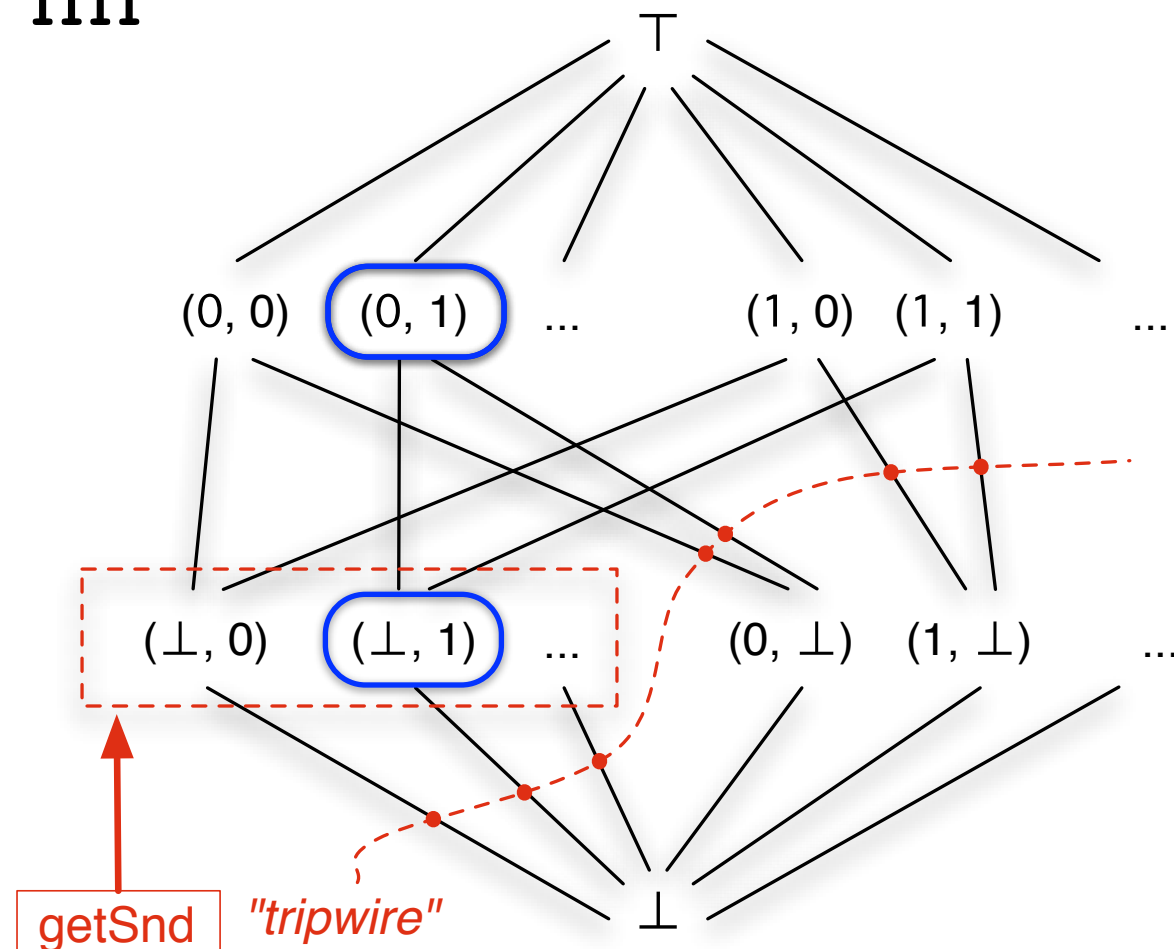


do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```

LVars: Threshold reads

nn



do

```
nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v -- returns 1
```

The threshold set must be *pairwise incompatible*

Monotonicity enables deterministic parallelism

INFORMATION PROCESSING 74 - NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type *o* (boolean, real, etc...) we could have declared a *o channel*. Then processes *f*, *g* and *h* are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes: at line (2) *f* is stated to communicate via two input lines that can carry integers, and one similar output line.

The body of a process is an usual Algol program except for invocation of *wait* on an input line (e.g. at (4)) or *send* a variable on a line of compatible type (e.g. at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.

In other words, processes communicate via first-in first-out (fifo) queues. Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes. Such a style of programming is close to many systems using EVENT mechanisms ([1],[2],[3],[4]). A pictorial representation of the program is the schema P on fig.2., where the nodes represent processes and the arcs communication channels between these processes.

What sort of things would we like to prove on a program like S? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole system would stop.

The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way: some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

```
Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
    Begin integer I ; Logical B ;
    B := true ;
    Repeat Begin
(4)      I := if B then wait(U) else wait(V) ;
(7)      print (I) ;
(5)      send I on W ;
          B := ¬B ;
    end ;
    End ;
Process g(integer in U ; integer out V, W) ;
Begin integer I ; Logical B ;
B := true ;
Repeat Begin
    I := wait (U) ;
    if B then send I on V else send I on W ;
    B := ¬B ;
End ;
End ;
(3) Process h(integer in U; integer out V; integer INIT);
Begin integer I ;
send INIT on V ;
Repeat Begin
    I := wait(U) ;
    send I on V ;
End ;
End ;
Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1)
End ;
```

Fig.1. Sample parallel program S.

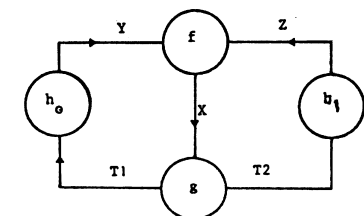


Fig.2. The schema P for the program S.

Monotonicity enables deterministic parallelism

f is *monotonic* iff, for a given \leq ,
$$x \leq y \implies f(x) \leq f(y)$$

INFORMATION PROCESSING 74 - NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type σ (boolean, real, etc...) we could have declared a σ channel. Then processes f , g and h are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes: at line (2) f is stated to communicate via two input lines that can carry integers, and one similar output line.

The body of a process is an usual Algol program except for invocation of *wait* on an input line (e.g. at (4)) or *send* a variable on a line of compatible type (e.g. at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.

In other words, processes communicate via first-in first-out (fifo) queues. Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes. Such a style of programming is close to many systems using EVENT mechanisms ([1],[2],[3],[4]). A pictorial representation of the program is the schema P on fig.2., where the nodes represent processes and the arcs communication channels between these processes.

What sort of things would we like to prove on a program like S? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole system would stop.

The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way: some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

```
Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
    Begin integer I ; Logical B ;
    B := true ;
    Repeat Begin
(4)      I := if B then wait(U) else wait(V) ;
(7)      print (I) ;
(5)      send I on W ;
          B := ¬B ;
    end ;
    End ;
Process g(integer in U ; integer out V, W) ;
Begin integer I ; Logical B ;
B := true ;
Repeat Begin
    I := wait (U) ;
    if B then send I on V else send I on W ;
    B := ¬B ;
End ;
End ;
(3) Process h(integer in U; integer out V; integer INIT);
Begin integer I ;
send INIT on V ;
Repeat Begin
    I := wait(U) ;
    send I on V ;
End ;
End ;
Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1)
End ;
```

Fig.1. Sample parallel program S.

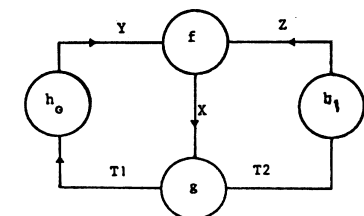


Fig.2. The schema P for the program S.

Kahn, 1974

Monotonicity enables deterministic parallelism

f is *monotonic* iff, for a given \leq ,
$$x \leq y \implies f(x) \leq f(y)$$

Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this is a crucial property since it allows parallel operation : a machine need not have all of its input to start computing, since future input concerns only future output.

INFORMATION PROCESSING 74 - NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. Its deficiencies are exhibited by this theoretical analysis. We hope in this way to make a case for languages for systems programming and

```
integer channel X, Y, Z, T1, T2 ;
process f(integer in U, V; integer out W) ;
  in integer I ; logical B ;
  B := true ;
  Repeat Begin
    I := if B then wait(U) else wait(V) ;
    print (I) ;
    send I on W ;
    B := not B ;
  end ;
;
process g(integer in U; integer out V, W) ;
  in integer I ; logical B ;
  B := true ;
  Repeat Begin
    I := wait (U) ;
    if B then send I on V else send I on W ;
    B := not B ;
  end ;
;
process h(integer in U; integer out V; integer INIT) ;
  in integer I ;
  and INIT on V ;
  Repeat Begin
    I := wait(U) ;
    send I on V ;
  end ;
;
Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1)
End ;
```

Fig.1. Sample parallel program S.

until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.
In other words, processes communicate via first-in first-out (fifo) queues.
Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* indicates the parallel composition of the processes.

2. PARALLEL COMPUTATION.
Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

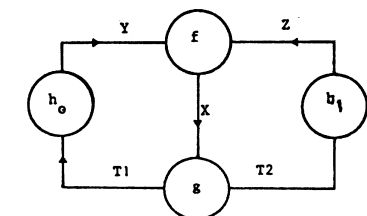


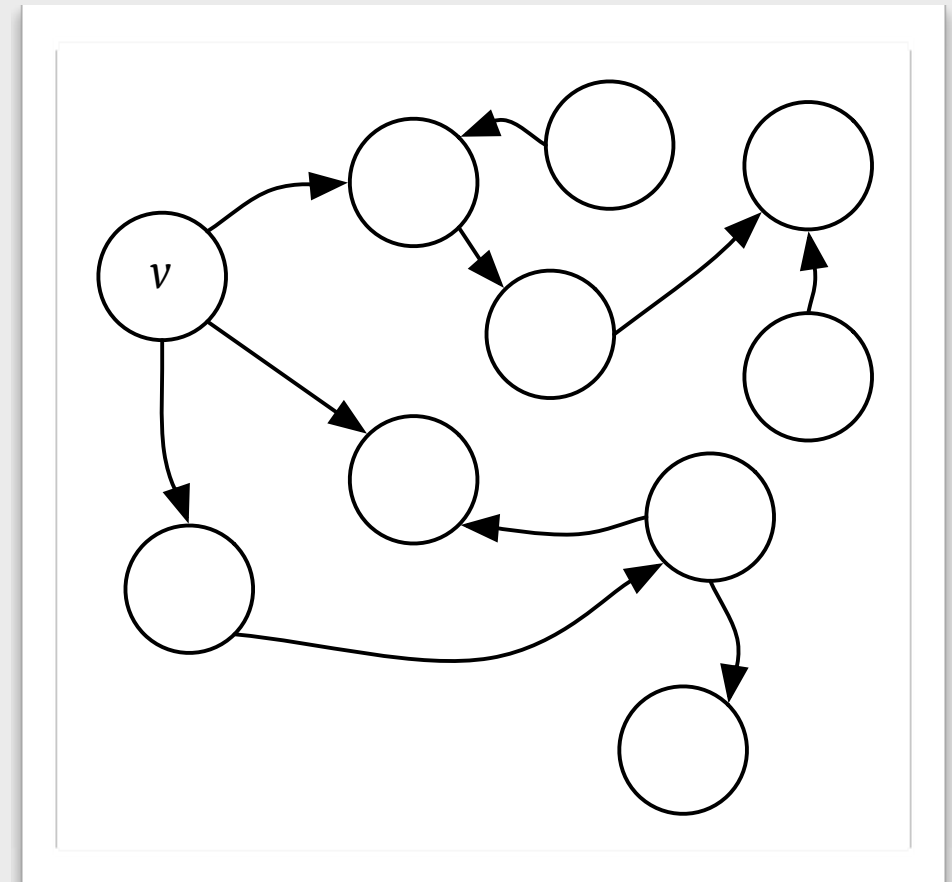
Fig.2. The schema P for the program S.

The kind of parallel programming we have studied in this paper is severely limited : it can produce only determinate programs.

Challenge problem

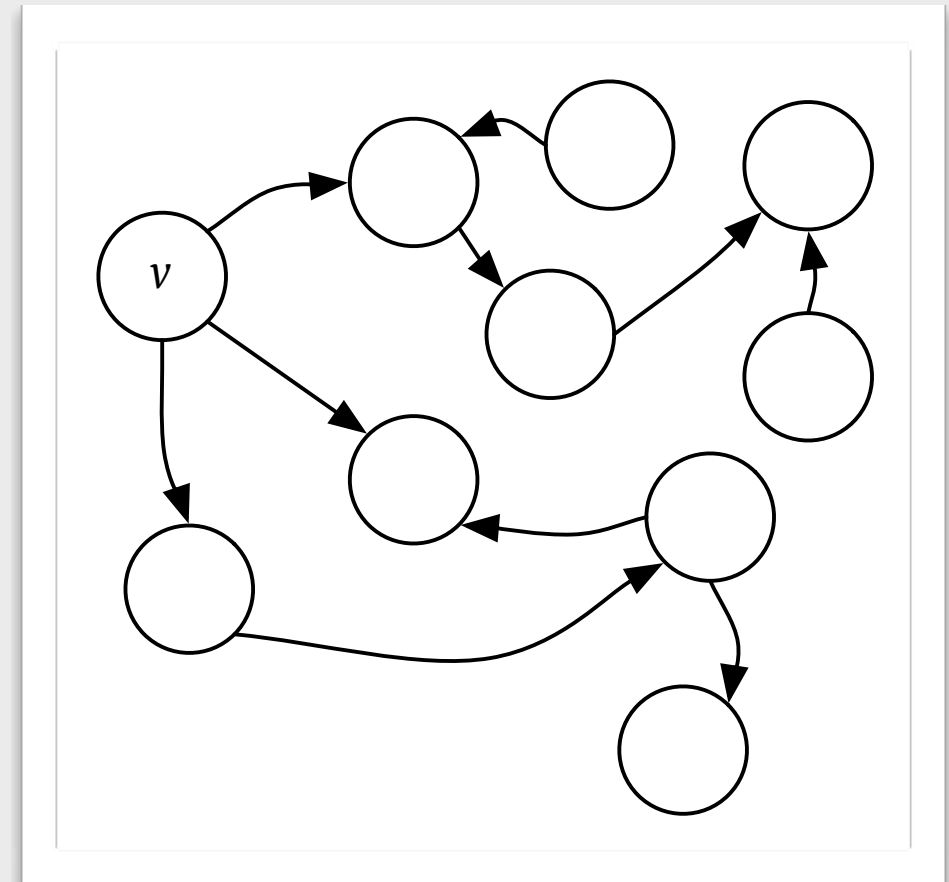
In a directed graph:

- find the connected component of all nodes within k hops of a vertex v
- and compute a function *analyze* over each vertex in that component
- **making the set of results available asynchronously to other computations**

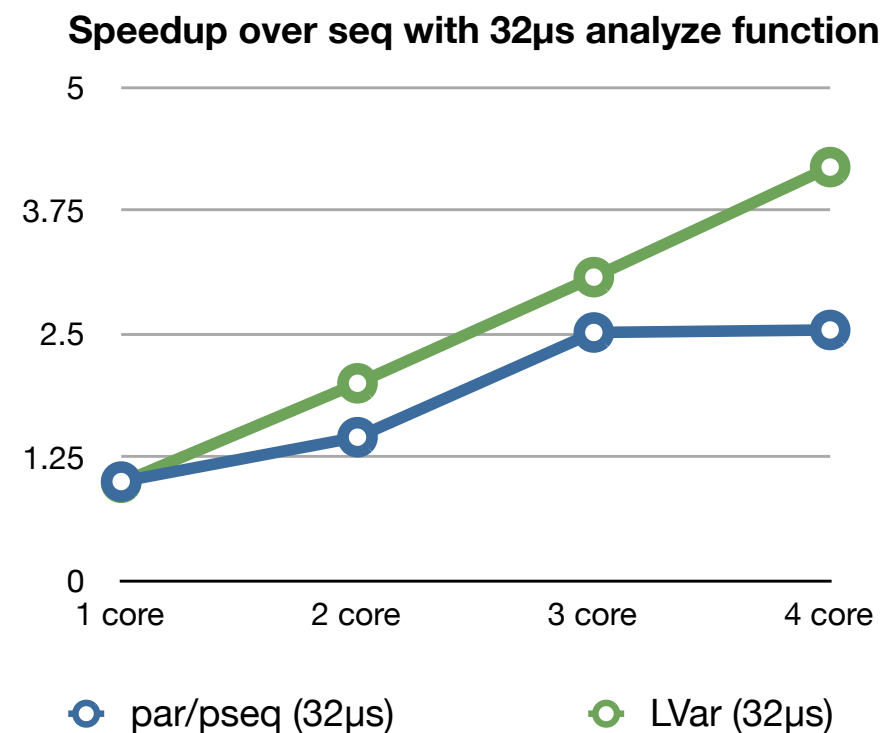
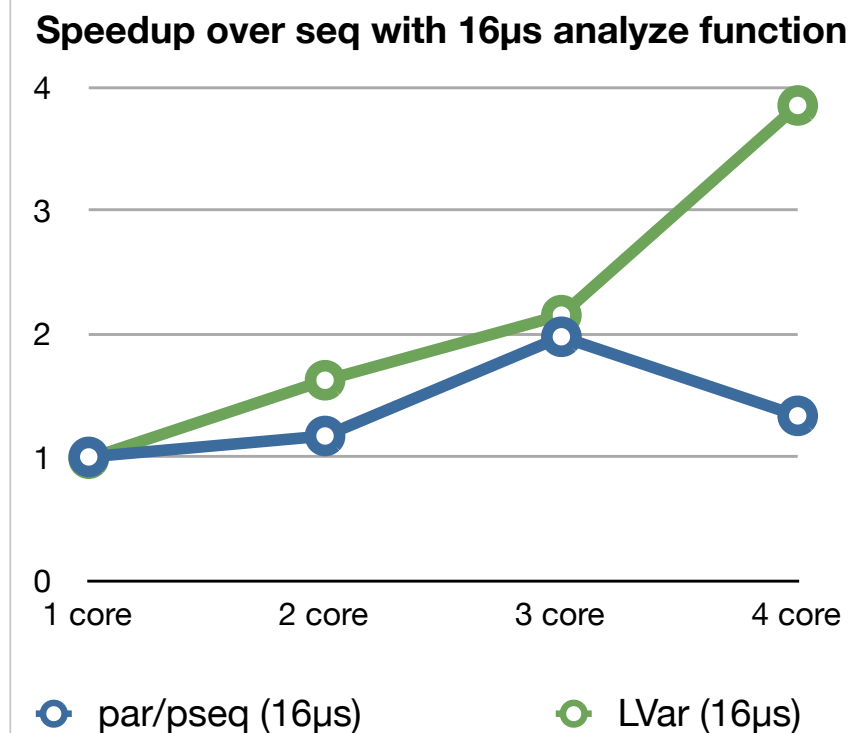
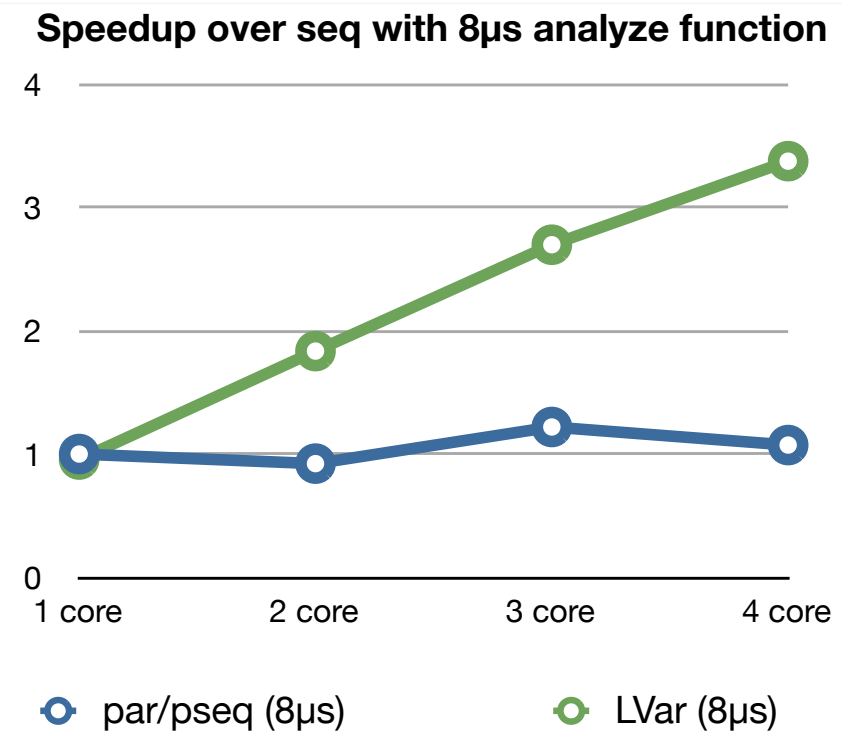
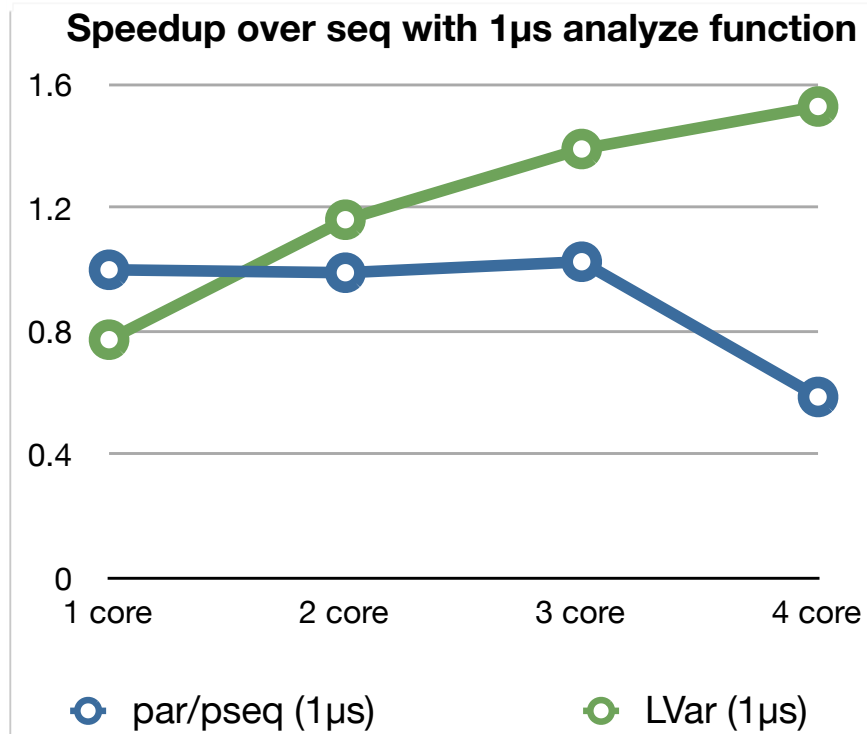


Challenge problem

- We compared two implementations:
 - `Control.Parallel.Strategies`
 - Our prototype LVar library (tracking visited nodes in an LVar)
- Level-sync breadth-first traversal, $k = 10$
- Random graph; 320K edges; 40K nodes
- Varying:
 - number of cores
 - amount of work done by *analyze*



Challenge problem: Strategies vs. LVars



Challenge problem: Strategies vs. LVars

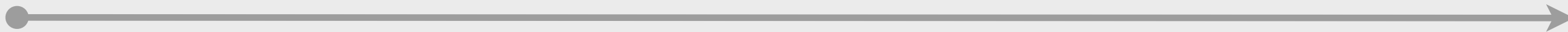
Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this is a crucial property since it allows parallel operation : a machine need not have all of its input to start computing, since future input concerns only future output.

- Average time from start of program to first invocation of *analyze*:
 - Strategies version: 64.64 ms
 - LVar version: 0.18 ms

(observably)

Deterministic Parallelism
via monotonic writes
and threshold reads.

July 2012



July 2012



July 2012



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, currency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, currency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter



<~ bloom



Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, currency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013



<~ bloom



Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, currency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013



<~ bloom



“It’s very
interesting stuff.”

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob’s removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is



“It’s very
interesting stuff.”

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob’s removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013

June 2013



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, currency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is



“It’s very interesting stuff.”

“I’d love to link a few co-workers to your blog post...”

July 2012

August 2012

long, dark winter

May 2013

June 2013

July 2013



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, currency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is



“It’s very
interesting stuff.”



FHPC '13

“I’d love to link a few co-workers
to your blog post...”



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway UC Berkeley nrc@cs.berkeley.edu
William R. Marczak UC Berkeley wrm@cs.berkeley.edu
Peter Alvaro UC Berkeley palvaro@cs.berkeley.edu
Joseph M. Hellerstein UC Berkeley hellerstein@cs.berkeley.edu
David Maier Portland State University maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the CALM theorem, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is



“It’s very interesting stuff.”



FHPC '13



POPL '14

“I’d love to link a few co-workers to your blog post...”

July 2012

August 2012

long, dark winter

May 2013

June 2013

July 2013

October 2013

RICON



<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, currency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is



“It’s very interesting stuff.”



FHPC '13



POPL '14



“I’d love to link a few co-workers to your blog post...”

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jambani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Conflict-Free

Marc Shapiro^{1,2}, Nuno Pregitzer¹

¹ CTR, Univ. of

² Univ. of

Abstract.

Replicating data across many replicas to accept updates improves performance and availability in distributed systems. However, publishing updates under a formal Strong Consistency conditions for consistency is called a Conflict-free Replicated Data Type (CRDT). Despite any number of limitations, CRDTs are guaranteed to be eventually consistent. We study a number of CRDTs, supporting both point and range queries, and develop large-scale distributed properties.

Keywords: Eventual Consistency, Distributed Systems

1 Introduction

Replication and consistency are such as the WWW, peer-to-peer, and other distributed systems. This constitutes a performance consistency conflict with availability. When network delays are large, disconnected operation consistency provides better availability at some replicas, while

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the CALM theorem, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination. In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules. In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic. In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

Abstract

Programs written in parallel composition are observable regardless of how hard to reproduce their behavior. Parallel software by construction is single-assignment and monotonically increasing. LVars ensure a "threshold" resource give a proof of language with respect to support a library but never wrong.

Categories and Features: current Programs; Definitions and Applications]; Con-

1. **Intro**
Programs written in Haskell are observable re-
hard-to-reproduce. The parallel software
language extension is programming
written using Haskell.
The most common by-constructive
meaning mathematical libraries and
libraries and libraries with fun-
Haskell programming.

Permission is granted to make copies of this document for non-profit educational use, provided that the copyright notice and the address of the author(s) must be included in the copy. For all other uses, permission should be sought from the author(s). For all other uses, permission should be sought from the author(s).

HPDC '13, Sep 2013
Copyright is held by ACM 978-1-4503-1000-0
<http://dx.doi.org/10.1145/2500000>

Abstract

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is to be useful within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place.

While much work has focused on identifying methods of deterministic parallel programming [5, 6, 13, 16, 17, 26], *guaranteed* determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that

POPL '14, January 22-24, 2014, San Diego, CA, USA.
Copyright is held by the owner/authors(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01...\$15.00.
<http://dx.doi.org/10.1145/2535838.2535842>

Big-2 test deterministic parallelism. Our goal is to create a broader, general-purpose deterministic-by-construction programming environment that can be used to build a wide range of applications. We seek an approach that is not tied to a particular data structure and that supports familiar notions from both functional and imperative programming. We want to be able to build a program that does, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. In other words, there are no deletions and no races. The program state, then, is a set that supports insertion but not removal, and that is commutative.

The *LVar* programming model recently proposed by Kuper and the author [15] is a simple, first-order, deterministic programming language for data structures [15]. In their model (which we review in Section 2), all shared data structures (called *LVars*) are monotonic, and the states that an *LVar* can take on form a *lattice*. Writes to an *LVar* are *commutative* in the sense that the order in which they are performed does not matter, and they monotonically increase the information in the *LVar*, and that they commute with one another. But commutativity does not mean that the order in which they are performed does not matter, whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the *LVar* model, the answer to the question “What is the value of *x*?” is not unique. It is a set of values, the *lattice value* of *x*. In any case, the reading thread will block until the *LVar* goes over a desired threshold. In a monotonic data structure, then, the information that is available to a thread is the information that was available at any time—but the presence of information is

Many algorithms are presented explicitly as fixpoints of monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vossahl and Werner Vogels

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vossahl and Werner Vogels

ABSTRACT

In recent years we have been interested in achieving application level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations, and to ensure that all operations are eventually reordered. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically generate a program module achieving consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of Bloom to support a wider range of operations and data types. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—desynchronization, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

bary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as group work [11, 37]. Shapiro, et. al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students to study teams. It uses two sets of *CRDTs*: one for Students and another for Teams. The application reads a version of Students and inserts the derived value into Teams. Conversely, Bob's removal is removed from Students by another application replica. The use of *CRDTs* ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived value is also removed. This is the *scope dilemma*: Bob's removal. This is outside the scope of *CRDT* operations.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large *CRDTs* (e.g., an eventually all consistent shopping cart) provide higher-level application semantics, but are more complex and their lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

Abstract

1. **Intro**
Programs with
parallel com-
observable re-
hard-to-repro-
parallel softw-
guage extensi-
programming
written using
The most
by-construction
meaning mat-
libraries and
mining with fun-
Haskell progr-

Permission to make
classroom use is
for profit or com-
on the first page.
author's(s) must be
republich, to post
and/or a fee. Reprint
FHPC '13, Sep.
Copyright is held
ACM 978-1-4503-
<http://dx.doi.org/10.1145/2488468>

Permission to make classroom use is free of profit or commercial sale. The author(s) must be credited on the first page. Republishing, posting, and/or a fee. Request permission from ACM. Copyright is held by ACM 978-1-4503-1111-1. <http://dx.doi.org/>

Abstract

Deterministic algorithms for program reproduction are proposed to develop a shared state semantics and an LVar takeoff with respect to that its content. Although it is not possible to extend "freeze" and "thaw" to add the events to be enabled. We prove that through freeze on every run error. We demonstrate a library for features, together with a model and a

Non-determinism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is to be useful, it must be possible to determine the program's execution path for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place.

One way to achieve this is by using the methods of deterministic parallel programming [5, 6, 13, 16], *guaranteed* determinism in real parallel programs remains a lofty and rarely achieved goal. A plausible alternative is to use the programmatic model of concurrent tasks that communicate in restricted ways that

Pemissions to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advertising and that copies bear this notice and the full citation information, including the ACM Copyright notice. For other copying, reprinting or translation, permission should be sought from ACM Publications Department. For more information, contact the ACM Publications Department, 1901 L Street, Suite 900, San Francisco, CA 94111, USA. Fax: +1 415 499 9732. Copyright 2006 ACM 978-1-59593-224-2/06/0004 ...\$5.00.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

POPF, 14e, January 22-24, 2014, San Diego, CA, USA.
Copyright is held by the author(s). All rights reserved. Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01...\$15.00.
<http://dx.doi.org/10.1145/2535838.2535842>

Neelakantan R. Krishnaswami
MPI-SWS
ryan.r.newton@cs.indiana.edu

prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level.

The simplest strategy is to allow no communication, forcing all parallel processes to be sequential. This is the approach taken by the parallel languages that follow this strategy [22], as the languages enforce force references to be either task-unique or immutable [5]. But this strategy is not the only one. For example, one can allow communication and force references, but restrict the way in which they are used. For example, one can restrict the way in which they are used to force state or message passing. A variety of deterministic programming construction models allow limited communication along these lines [1, 2, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voss
and Werner Vogels

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations, such as Bloom filters, that can be applied in any ordering. A more powerful approach was recently captured by the *CALM* theorem, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically generate programs that achieve consistency without coordination.

In this paper we present Bloom⁺, an extension to Bloom that takes inspiration from both these contrasting Bloom⁺ generators. We extend Bloom⁺ to support a rich set of operations. CALM analysis to whole programs remains amenable to lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code used in Bloom⁺. We show how Bloom⁺ can be extended to support how we use Bloom⁺ to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom⁺ encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other consistency-critical operations are implemented in the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of a consistent data management library that merges conflicting updates for these techniques have received significant attention in recent research. *Convergent Models and Monotonic Logic*.

Convergent Models: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message ordering and reply. For example, a module may promise to return the same value for updates to data items in a key-value store; the user of the li-

rary need only restrict commutativity, associativity, idempotence and membership functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 39] as well as group theory [20]. The approach is based on the idea of *CRDTs* (Conflict-free Replicated Data Types) (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs are defined as (a) the program *replicas* that are responsible for ensuring lattice properties for the methods (commutativity, associativity, idempotence), and (b) *CRDTs* only provide guarantees for individual data objects (not sets of objects) [34, 35]. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant courseware application assigns students to study teams. It uses two sets of CRDTs: one for Students and another for Teams. The application has two replicas. Replica 1 has a student named *element* *«Alice.Bob»* in *Teams*. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree on the state of the application. However, the application-level state is inconsistent unless the derived value in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules and the *simple semantic dilemma* of a module (e.g., a set) making a *simple semantic decision* (e.g., inspecting a set for a particular element) are not solvable in general. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties for the application. This is often difficult because of the difficulty to test, maintain, and trust.

MONOTONIC LOGIC. In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency [15, 18, 25]. Monotonic logic is a logic that never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *conflict* (invariant) free [15, 18, 25]. Monotonicity of a Datalog program is consistent [5, 18, 25].

Abstract

Programs written in parallel composition are observable regardless of how hard-to-reproduce parallel software is constructed by construction. This single-assignment monotonicity theorem ensures that LVars ensure a "threshold" reduction give a proof of language with respect to support a library but never wrong.

Categories and Features: current Program Definitions and Applications]; Con-

Keywords D

1. Intro

Programs written in parallel computing languages are observable re-usable, hard-to-reproduce, parallel software. Language extensions for parallel programming, written using Haskell.

The most common by-construction meaning matrix libraries and running with Haskell programs.

Permission to make classroom use is granted for profit or commercial use on the first page. The author(s) must be acknowledged, to post online and/or a fee. Request FHPAC '13, September. Copyright is held by ACM 978-1-4503-1140-1/13/09/0000. <http://dx.doi.org/10.1145/2474641.2474642>

[illegible]

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is *observable* within a program, it becomes much more difficult for programmers to design correct and correct bys testing, let alone to reason about their programs in the first place.

While much work has focused on identifying models of deterministic parallel programming [5, 6, 13, 16, 17, 26], *guaranteed* determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from www.acm.org/permissions.

POPE, '14, January 22–24, 2014, San Diego, CA, USA.
Copyright is held by the owner(s) of the Publication Rights and licensed to ACM.
ACM 978-1-4503-2544-8/14/01...\$15.00.
<http://dx.doi.org/10.1145/2535838.2535842>

The simplest strategy is to allow no communication, forcing all concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [22], as do languages that force references to be either task-unique or immutable [5]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction languages support communication along fixed lines, but they tend to be narrow in scope and centered around a single data structure: for instance, FIFO queues in Kahn process networks [13] and StreamIt [11], or shared write-only tables in Intel Concurrency Collections [6].

General purpose deterministic parallelism Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We have implemented a number of features that we think will be useful to those who support familiar idioms from both functional and imperative programming languages. Our starting point is the idea of monads and impure computations. We have implemented (1) a type system for monads, (2) a set of familiar monadic operations, and (3) the notion that information is added is not observable. A paradigmatic example is a set that supports insertion but not deletion. The *LX* programming model recently proposed by Kuper and Shiple [15] is a good example of a language that implements this data structure [15]. In their model, we have (see Section 2.2.1) all shared data structures in *LX* programs are monotonic, and the states that an *LX* can take on form a lattice. We have also implemented a number of features that we think will be useful to those who think that they monotonically increase the information in the *LX* program. We have implemented (1) a type system for information, (2) a set of familiar operations for information, and (3) a notion that information is not enough to guarantee determinism. If a read can observe whether or not a concurrent write has happened, then it can observe determinism in scheduling. So, in the *LX* model, the answer to the question of whether or not a read can observe determinism is that it can observe determinism in the lattice values it always sees; the reading thread will block until it sees the lattice value it wants. This is a good idea because it allows the absence of information to be transient. In another thread could add that information at any times that the presence of information could

The LVars model guarantees determinism, supports an unlimited variety of data structures (anything viewable as a lattice), and provides a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as one might hope.

Many algorithms are presented explicitly as fixpoints of monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it

Conflict-Free

Marc Shapiro^{1,2}, Nuno Freire

Abstract. Replicating in any replica to accept updates means performance and security trade-offs. Under a formal Strong Consistency condition for replicated conditions is called a Conflict-free Replicated Data Type (CRDT). We study a number of CRDTs (state- and operation-based), supporting both depth and the more complex G-conditions. We develop large-scale distributed algorithms for CRDTs.

Keywords: Eventual Consistency Distributed Systems

When network delays are low, network consistency promises better accuracy at some replica, without

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vossshall and Werner Vogels

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years we have been interested in achieving application level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations, and to ensure that all operations are eventually reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically generate a program module achieving consistency without coordination.

In this paper we present Bloom², an extension to Bloom¹ that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of Bloom to support a wider range of operations and data structures. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

rary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1 A replicated, fault-tolerant courseware application assigns students into study teams. It uses two sets of CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived value in Teams is updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

LVars: Lattice-based Data Structures for Deterministic Parallelism

Abstract
Programs written using a parallel computation are observable results, offering hard-to-reproduce nondeterministic parallel software. We present by-construction parallel programming models that ensure deterministic results. LVars ensure deterministic results by giving a proof of deterministic language with LVars and data to support a limited form, but never wrong answers.

Categories and Subject Descriptors
Current Programming Languages; Concurrent Programming; Definitions and Theories; Concurrent, distributed computing

Lindsey Kuper
Indiana University
lkuper@cs.indiana.edu

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Neelakantan R. Krishnaswami
MPI-SWS
neel@mpi-sws.org

Abstract

Deterministic-by-construction parallel programming models offer programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. A principled approach to deterministic-by-construction parallel programming with shared state is offered by LVars: shared memory locations whose semantics are defined in terms of a user-specified lattice. Writes to an LVar take the least upper bound of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. Although it guarantees determinism, this interface is quite limited. We extend LVars in two ways. First, we add the ability to “freeze” and then read the contents of an LVar directly. Second, we add the ability to attach callback functions to an LVar, allowing events to be triggered by writes to it. Together, callbacks and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through freezable LVars, programs are at worst quasi-deterministic: on every run, they either produce the same answer or raise an error. We demonstrate the viability of our approach by implementing a library for Haskell supporting a variety of LVar-based data structures, together with two case studies that illustrate the programming model and yield promising parallel speedup.

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is observable within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place. While much work has focused on identifying methods of deterministic parallel programming [5, 6, 13, 16, 17, 26], guaranteed determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that

prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level. The simplest strategy is to allow no communication, forcing concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [22], as do languages that force references to be either task-unique or immutable [5]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction models allow limited communication along these lines, but they tend to be narrow in scope and centered around a single data structure: for instance, FIFO queues in Kahn process networks [13] and Streams [11], or shared write-only tables in Intel Concurrent Collections [6].

Big-ten deterministic parallelism. Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We seek an approach that is not tied to a particular data structure and that supports familiar idioms from both functional and imperative languages. Our starting point is the idea of *monotonic* data structures, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others.

The LVars programming model recently proposed by Kuper and Newton makes an initial foray into programming with monotonic data structures [15]. In their model (which we review in Section 2), all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a *lattice*. Writes to an LVar must correspond to a join (least upper bound) in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the LVar model, the answer to the question “has a write occurred?” (i.e., is the LVar above a certain lattice value?) is always *yes*; the reading thread will block until the LVar goes over a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.

The LVars model guarantees deterministic results by providing a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as we might hope. Many algorithms are presented explicitly as monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of “seen nodes”; neighbors of seen nodes are fed back into the set until it

Freeze After Writing

Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper
Indiana University
lkuper@cs.indiana.edu

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Neelakantan R. Krishnaswami
MPI-SWS
neel@mpi-sws.org

Ryan R. Newton
Indiana University
rnewton@cs.indiana.edu

Abstract

Deterministic-by-construction parallel programming models offer programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. A principled approach to deterministic-by-construction parallel programming with shared state is offered by LVars: shared memory locations whose semantics are defined in terms of a user-specified lattice. Writes to an LVar take the least upper bound of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. Although it guarantees determinism, this interface is quite limited. We extend LVars in two ways. First, we add the ability to “freeze” and then read the contents of an LVar directly. Second, we add the ability to attach callback functions to an LVar, allowing events to be triggered by writes to it. Together, callbacks and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through freezable LVars, programs are at worst quasi-deterministic: on every run, they either produce the same answer or raise an error. We demonstrate the viability of our approach by implementing a library for Haskell supporting a variety of LVar-based data structures, together with two case studies that illustrate the programming model and yield promising parallel speedup.

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is observable within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place. While much work has focused on identifying methods of deterministic parallel programming [5, 6, 13, 16, 17, 26], guaranteed determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that

prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level. The simplest strategy is to allow no communication, forcing concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [22], as do languages that force references to be either task-unique or immutable [5]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction models allow limited communication along these lines, but they tend to be narrow in scope and centered around a single data structure: for instance, FIFO queues in Kahn process networks [13] and Streams [11], or shared write-only tables in Intel Concurrent Collections [6].

Big-ten deterministic parallelism. Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We seek an approach that is not tied to a particular data structure and that supports familiar idioms from both functional and imperative languages. Our starting point is the idea of *monotonic* data structures, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others.

The LVars programming model recently proposed by Kuper and Newton makes an initial foray into programming with monotonic data structures [15]. In their model (which we review in Section 2), all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a *lattice*. Writes to an LVar must correspond to a join (least upper bound) in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the LVar model, the answer to the question “has a write occurred?” (i.e., is the LVar above a certain lattice value?) is always *yes*; the reading thread will block until the LVar goes over a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.

The LVars model guarantees deterministic results by providing a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as we might hope. Many algorithms are presented explicitly as monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of “seen nodes”; neighbors of seen nodes are fed back into the set until it

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jambani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels
Amazon.com

Conflict-Free Replicated Data Types*

Marc Shapiro^{1,2}, Nuno Preguiça^{1,2}, Carlos Baquero³, and Marek Żukowski⁴

¹ ISIRIA, Paris, France
² CITIL, Universidade Nova de Lisboa, Portugal
³ Universidade do Minho, Portugal
⁴ LIPN, Paris, France

Abstract. Replicating data under *Eventual Consistency* (EC) allows any replica to accept updates without requiring synchronization. This means performance and scalability in large-scale distributed systems (e.g., cloud). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a *Conflict-free Replicated Data Type* (CRDT). Replicas of any CRDT are guaranteed to converge to a self-stabilizing state despite any number of failures. This paper describes two popular protocols (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with element deletion, supporting both add and remove operations, and consider the more complex Graph data type. CRDT types can be compared to develop large-scale distributed applications, and have interesting central properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system such as the WWW, peer-to-peer, or cloud computing platforms. The “strong consistency” approach serializes updates in a global total order. This constitutes a performance and scalability bottleneck. Furthermore, consistency conflicts with availability and partition-tolerance [2].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [1, 12]. An update executes at some replica, without synchronization; later, it is sent to the other

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules. In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant coursework application assigns students into study teams. It uses two sets of CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a *scope dilemma*: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic. In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

1

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jambani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels

Conflict-Free

Marc Shapiro^{1,2}, Nuno Preguiça^{1,2}, Carlos Baquero³, and Marek Żukowski⁴

¹ ISIRIA, Paris, France
² CITIL, Universidade Nova de Lisboa, Portugal
³ Universidade do Minho, Portugal
⁴ LIPN, Paris, France

Abstract. Replicating data under *Eventual Consistency* (EC) allows any replica to accept updates without requiring synchronization. This means performance and scalability in large-scale distributed systems (e.g., cloud). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a *Conflict-free Replicated Data Type* (CRDT). Replicas of any CRDT are guaranteed to converge to a self-stabilizing state despite any number of failures. This paper describes two popular protocols (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with element deletion, supporting both add and remove operations, and consider the more complex Graph data type. CRDT types can be compared to develop large-scale distributed applications, and have interesting central properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system such as the WWW, peer-to-peer, or cloud computing platforms. The “strong consistency” approach serializes updates in a global total order. This constitutes a performance and scalability bottleneck. Furthermore, consistency conflicts with availability and partition-tolerance [2].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [1, 12]. An update executes at some replica, without synchronization; later, it is sent to the other

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom², an extension to Bloom that takes inspiration from both these traditions. Bloom² generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom² to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom² encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules. In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the li-

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. A replicated, fault-tolerant coursework application assigns students into study teams. It uses two sets of CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.

Taken together, the problems with Convergent Modules present a *scope dilemma*: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic. In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never “retracts” an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

1

Thank you!

Email: lkuper@cs.indiana.edu

Research blog: composition.al

Project repo: github.com/iu-parfunc/lvars

Code from this talk: github.com/lkuper/lvar-examples

Special thanks to: Ryan Newton, Aaron Turon, Neel Krishnaswami, Sam Tobin-Hochstadt, Amr Sabry, Vincent St-Amour, Neil Conway, Sam Elliott, Mike Bernstein, and the IU PL Wonks.