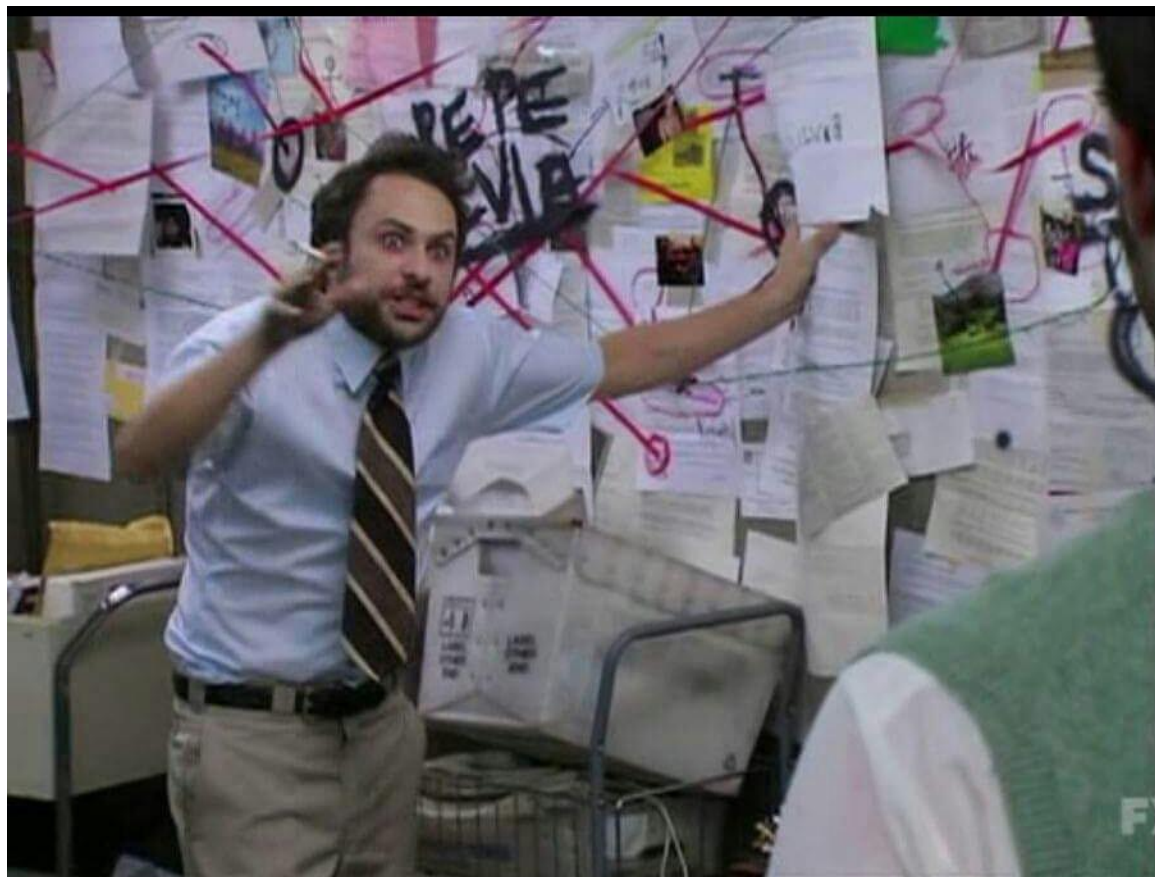


Keep CALM and CRDT On

[Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein.](#)
[Keep CALM and CRDT On, PVLDB, 16\(4\): 856 - 863, 2022](#)

PWL Bangalore – 19th Oct. 2023



```
$ whoami
```

What is coordination?

“Knowledge Is The Dual of Possibility.”

Knowledge and Common Knowledge
in a Distributed Environment*

[J. Halpern et al. *Knowledge and Common Knowledge In A Distributed Environment*](#)

Joseph Y. Halpern

Yoram Moses

IBM Almaden Research Center
San Jose, CA 95120

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, 76100 ISRAEL

Abstract: Reasoning about knowledge seems to play a fundamental role in distributed systems. Indeed, such reasoning is a central part of the informal intuitive arguments used in the design of distributed protocols. Communication in a distributed system can be viewed as the act of transforming the system's state of knowledge. This paper presents a general framework for formalizing and reasoning about knowledge in distributed systems. We argue that states of knowledge of groups of processors are useful concepts for the design and analysis of distributed protocols. In particular, *distributed knowledge* corresponds to knowledge that is “distributed” among the members of the group, while *common knowledge* corresponds to a fact being “publicly known”. The relationship between common knowledge and a variety of desirable actions in a distributed system is illustrated. Furthermore, it is shown that, formally speaking, in practical systems common knowledge cannot be attained. A number of weaker variants of common knowledge that are attainable in many cases of interest are introduced and investigated.

What is coordination?

“Knowledge Is The Dual of Possibility.”

Knowledge and Common Knowledge
in a Distributed Environment*

Joseph Y. Halpern

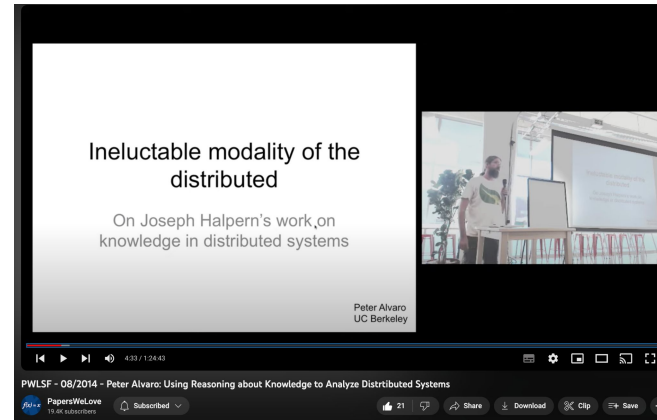
Yoram Moses

IBM Almaden Research Center
San Jose, CA 95120

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, 76100 ISRAEL

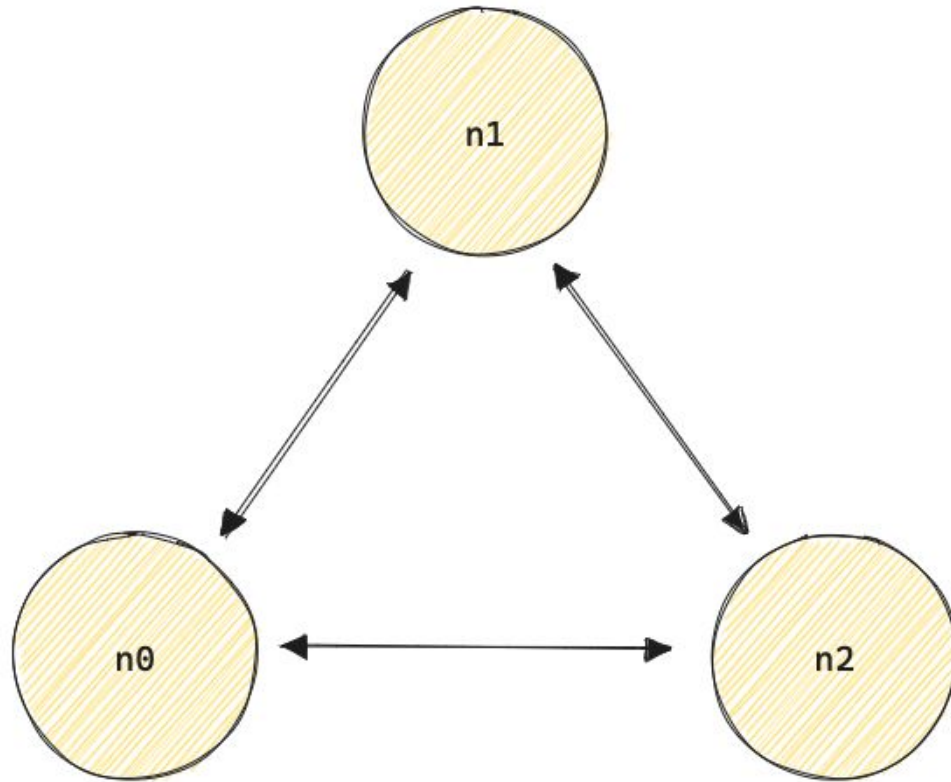
Abstract: Reasoning about knowledge seems to play a fundamental role in distributed systems. Indeed, such reasoning is a central part of the informal intuitive arguments used in the design of distributed protocols. Communication in a distributed system can be viewed as the act of transforming the system's state of knowledge. This paper presents a general framework for formalizing and reasoning about knowledge in distributed systems. We argue that states of knowledge of groups of processors are useful concepts for the design and analysis of distributed protocols. In particular, *distributed knowledge* corresponds to knowledge that is “distributed” among the members of the group, while *common knowledge* corresponds to a fact being “publicly known”. The relationship between common knowledge and a variety of desirable actions in a distributed system is illustrated. Furthermore, it is shown that, formally speaking, in practical systems common knowledge cannot be attained. A number of weaker variants of common knowledge that are attainable in many cases of interest are introduced and investigated.

[J. Halpern et al. *Knowledge and Common Knowledge In A Distributed Environment*](#)

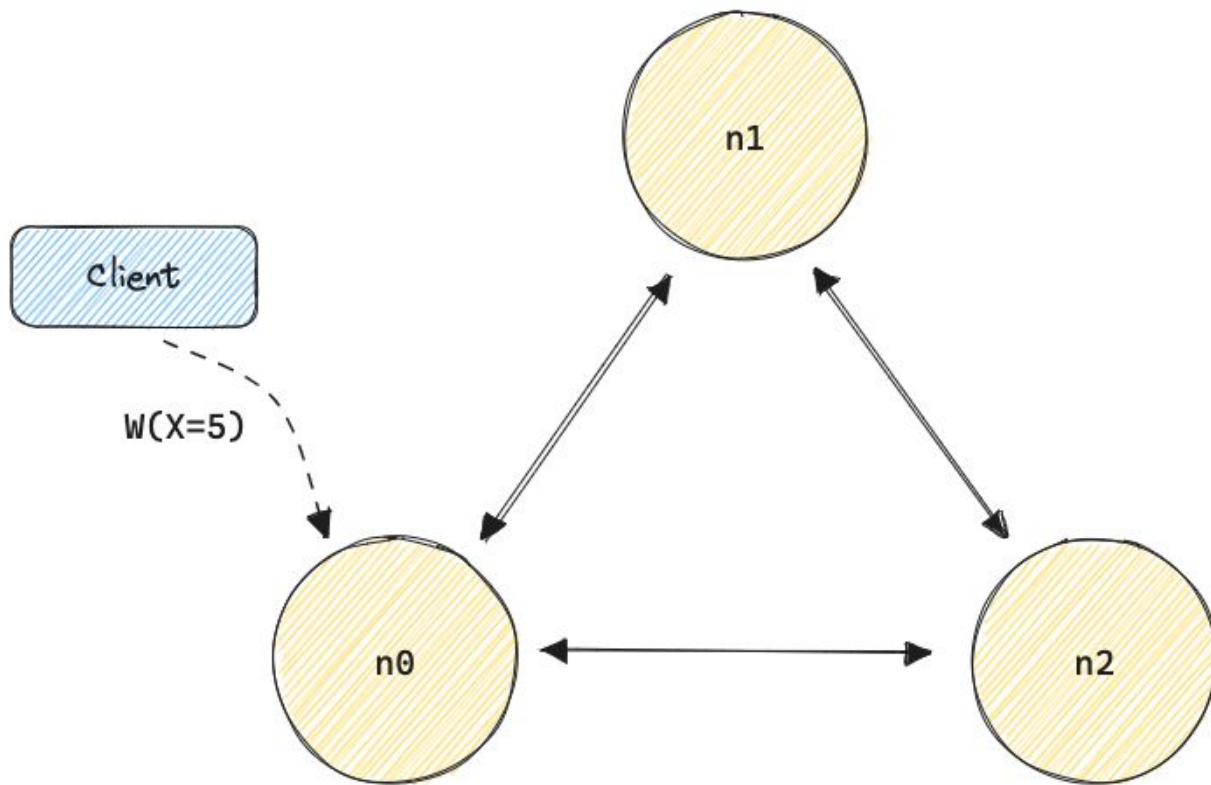


<https://www.youtube.com/watch?v=7U0qPmEpbSI&list=WL&index=21>

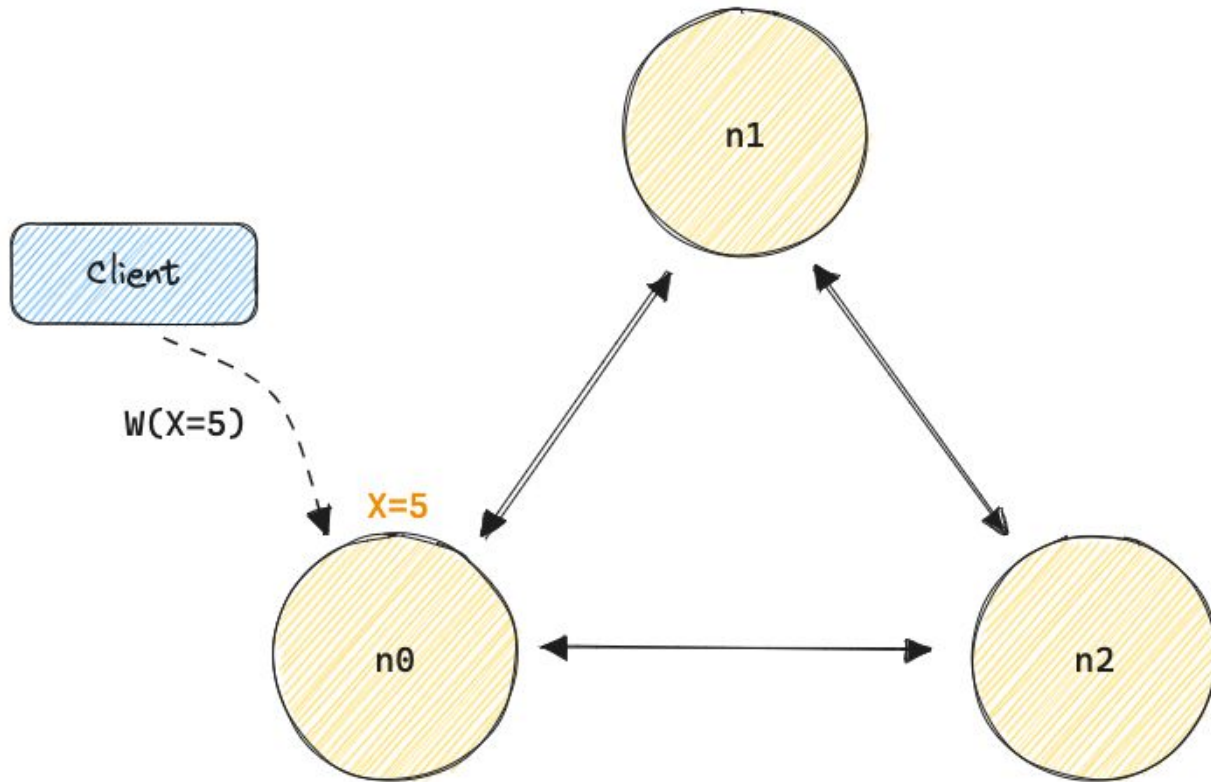
What is coordination?



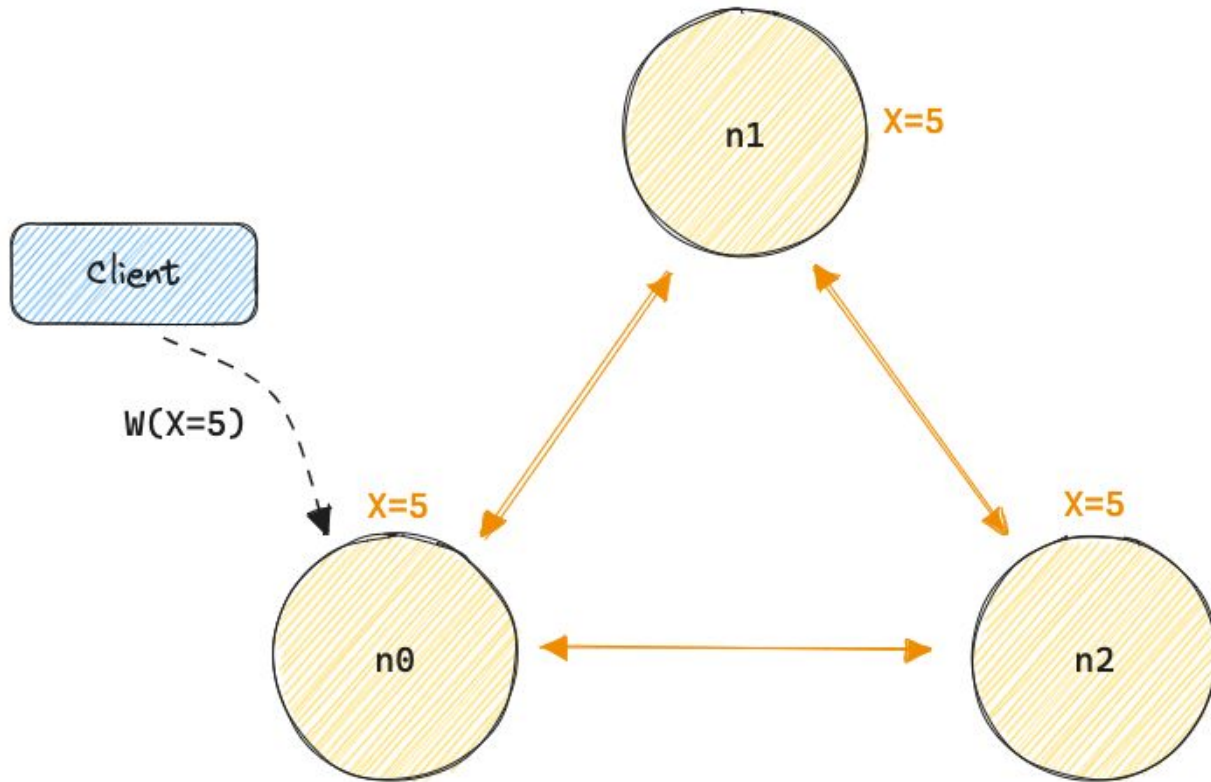
What is coordination?



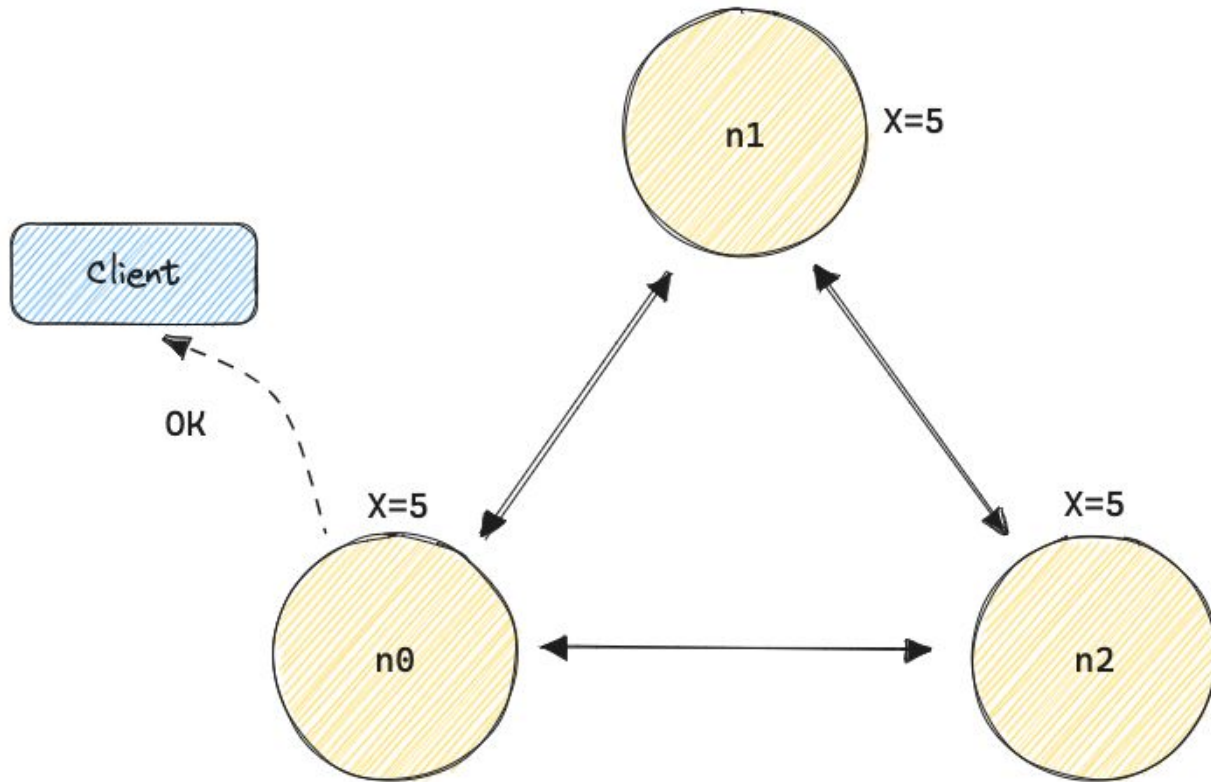
What is coordination?



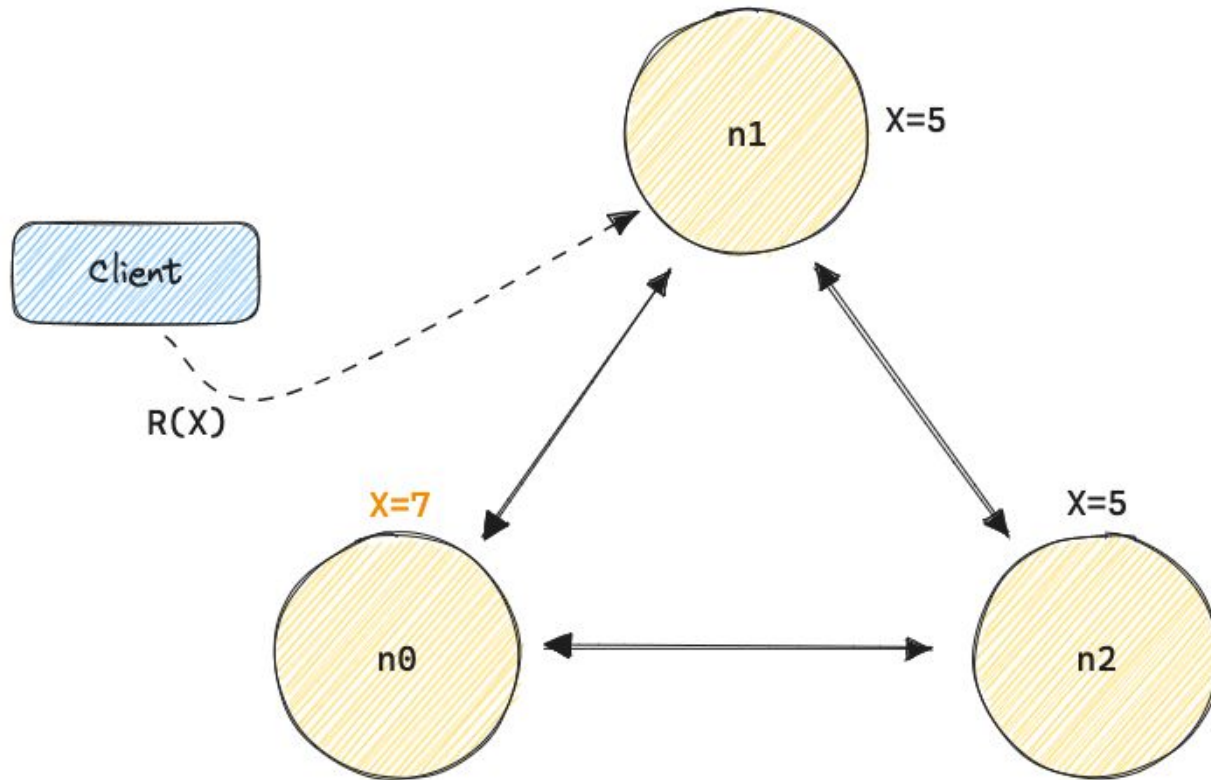
What is coordination?



What is coordination?



What is coordination?



What is coordination?

There's also message re-ordering, network partitions and all other flavours of why distributed systems are hard.

What is coordination?

We also have other coordination mechanisms like 2PC.

What is coordination?

- In any case, coordination mechanisms are a way to synchronize access to a shared memory of some sort.
- They are probably the most well studied class of algorithms in Distributed Systems literature.

Downside of coordination

Coordination mechanisms have massive performance costs attached to them.

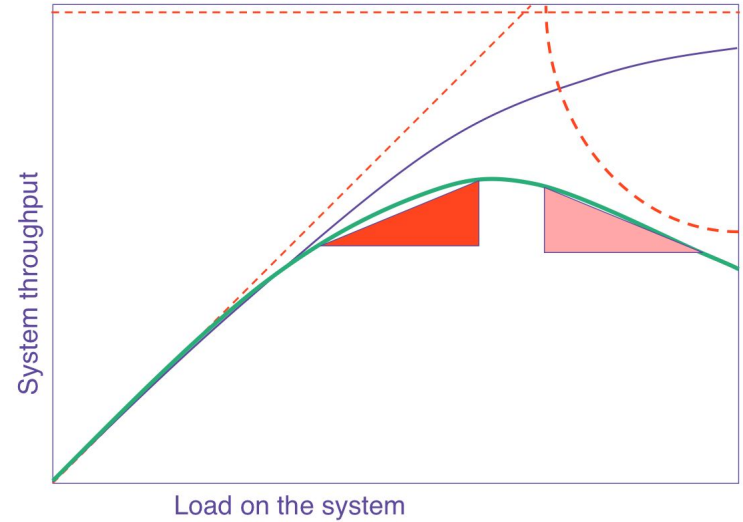
“The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them”

James Hamilton, SVP and Distinguished Engineer at AWS

Downside of coordination

Intuition from Universal Scalability Law (USL).

- Linear scalability is a sham.
- As work done to achieve data consistency (“coherency”) increases, it starts to bottleneck your system’s throughput.



<http://www.perfdynamics.com/Manifesto/USLscalability.html>

Downside of coordination

Coordination Avoidance in Database Systems

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

ABSTRACT

Minimizing coordination, or blocking communication between concurrently executing operations, is key to maximizing scalability, availability, and high performance in database systems. However, uninhibited coordination-free execution can compromise application correctness, or consistency. When is coordination necessary for correctness? The classic use of serializable transactions is sufficient to maintain correctness but is not necessary for all applications, sacrificing potential scalability. In this paper, we develop a formal framework, invariant confluence, that determines whether an application requires coordination for correct execution. By operating on application-level invariants over database states (e.g., integrity constraints), invariant confluence analysis provides a necessary and sufficient condition for safe, coordination-free execution. When programmers specify their application invariants, this analysis allows databases to coordinate only when anomalies that might violate invariants are possible. We analyze the invariant confluence of common invariants and operations from real-world database systems (i.e., integrity constraints) and applications and show that many are invariant confluent and therefore achievable without coordination. We apply these results to a proof-of-concept coordination-avoiding database prototype and demonstrate sizable performance gains compared to serializable execution, notably a 25-fold improvement over prior TPC-C New-Order performance on a 200 server cluster.

level correctness, or consistency.¹ In canonical examples, concurrent, coordination-free work can result in undesirable and “inconsistent” outcomes that account for application-level anomalies that should be prevented. To ensure correct behavior, a database must coordinate the execution of these operations that are executed concurrently, could result in inconsistent

This tension between coordination and correctness by the range of database concurrency control in traditional database systems, serializable isolation of operations (transactions) with the illusion of serial order [15]. As long as individual transaction application state, serializability guarantees that, however, each pair of concurrent operations (at least a write) can potentially compromise serializability and will require coordination to execute [9, 21]. Because the level of reads and writes, serializability is a conservative invariant and may in turn coordinate more than is necessary for consistency [29, 39, 53, 58]. For example, a database can safely and simultaneously retweet Barack without observing a serial ordering of updates to InnoDB. In contrast, a range of widely-deployed weaker coordination to execute but surface read and write in turn compromise consistency [2, 9, 22, 48]. In models, it is up to users to decide when weak

Anna: A KVS For Any Scale

Chenggang Wu ^{#1}, Jose M. Faleiro ^{*2}, Yihan Lin ^{**3}, Joseph M. Hellerstein ^{#4}

[#] UC Berkeley

USA

¹ cgwu@cs.berkeley.edu

⁴ hellerstein@berkeley.edu

^{*} Yale University

USA

² jose.faleiro@yale.edu

^{**} Columbia University

USA

³ yihan.lin@columbia.edu

Abstract—Modern cloud providers offer dense hardware with multiple cores and large memories, hosted in global platforms. This raises the challenge of implementing high-performance software systems that can effectively scale from a single core to multicore to the globe. Conventional wisdom says that software designed for one scale point needs to be rewritten when scaling up by $10 - 100 \times$ [1]. In contrast, we explore how a system can be architected to scale across many orders of magnitude by design.

We explore this challenge in the context of a new key-value store system called Anna: a partitioned, multi-mastered system that achieves high performance and elasticity via wait-free execution and coordination-free consistency. Our design rests on a simple architecture of coordination-free actors that perform state update via merge of lattice-based composite data structures. We demonstrate that a wide variety of consistency models can be elegantly implemented in this architecture with unprecedented consistency, smooth fine-grained elasticity, and performance that far exceeds the state of the art.

also across cores for high performance. Second, to enable workload scaling, we need to employ **multi-master replication** to concurrently serve puts and gets against a single key from multiple threads.

The next two design requirements followed from our ambitions for performance and generality. To achieve maximum hardware utilization and performance within a multi-core machine, our third requirement was to guarantee **wait-free execution**, meaning that each thread is always doing useful work (serving requests), and never waiting for other threads for reasons of consistency or semantics. To that end, coordination techniques such as locking, consensus protocols or even “lock-free” retries [7] need to be avoided. Finally, to support a wide range of application semantics without compromising our other goals, we require a unified implementation for a

Can we avoid coordination?

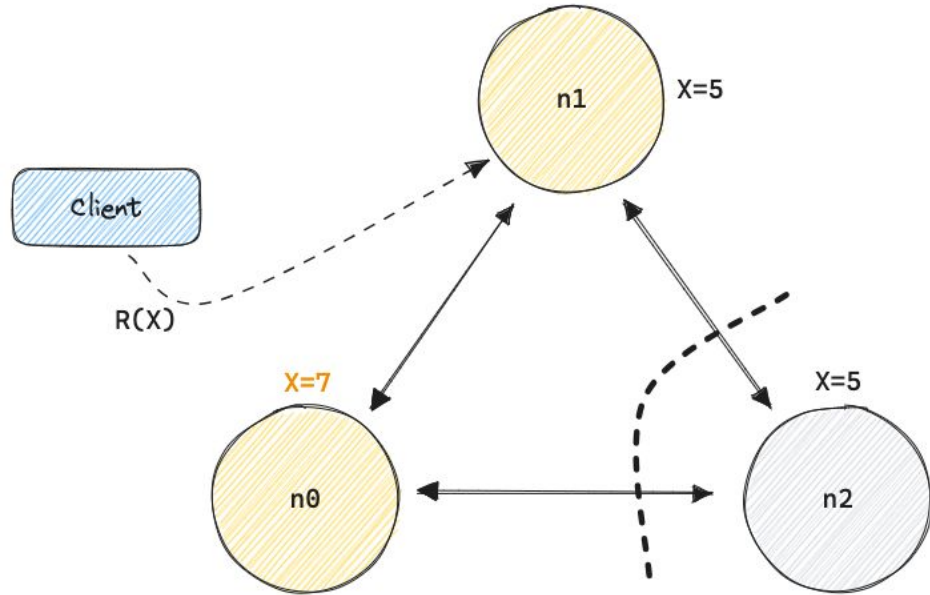
Can we avoid coordination?

“The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them”

James Hamilton, SVP and Distinguished Engineer at AWS

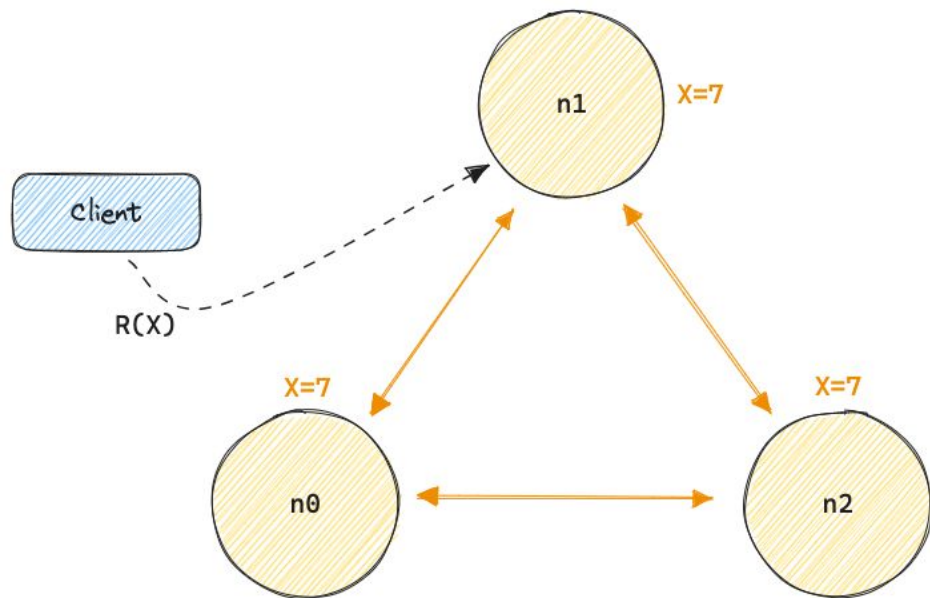
Can we avoid coordination?

A significant amount of non-determinism exists in distributed systems – uncoordinated parallel execution on unreliable machines, message order delivery, network failures, network partitions etc.



Can we avoid coordination?

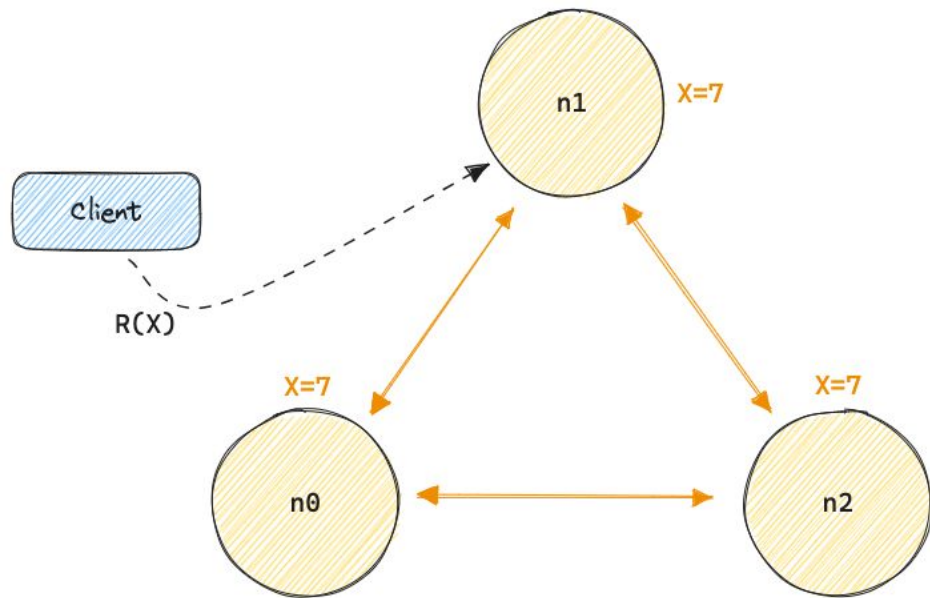
In an attempt to tame this non-determinism, we try and coordinate, we try and accumulate as much knowledge as possible about what the global state of the system might look like, and then take an action based on that.



Can we avoid coordination?

We coordinate in hopes of providing some guarantees for our system, guarantees which can be bucketed broadly as:

- Recency guarantees (ex: linearizability)
- Ordering guarantees (ex: sequential consistency, serializability).



Can we avoid coordination?

One way of avoiding coordination in transactional database systems is using invariants. If a *local* transaction can be shown to not violate a *global* invariant, we can avoid coordinating on this transaction.

Invariant confluence.

Coordination Avoidance in Database Systems

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

ABSTRACT

Minimizing coordination, or blocking communication between concurrently executing operations, is key to maximizing scalability, availability, and high performance in database systems. However, uninhibited coordination-free execution can compromise application correctness, or consistency. When is coordination necessary for correctness? The classic use of serializable transactions is sufficient to maintain correctness but is not necessary for all applications, sacrificing potential scalability. In this paper, we develop a formal framework, invariant confluence, that determines whether an application requires coordination for correct execution. By operating on application-level invariants over database states (e.g., integrity constraints), invariant confluence analysis provides a necessary and sufficient condition for safe, coordination-free execution. When programmers specify their application invariants, this analysis allows databases to coordinate only when anomalies that might violate invariants are possible. We analyze the invariant confluence of common invariants and operations from real-world database systems (i.e., integrity constraints) and applications and show that many are invariant confluent and therefore achievable without coordination. We apply these results to a proof-of-concept coordination-avoiding database prototype and demonstrate sizable performance gains compared to serializable execution, notably a 25-fold improvement over prior TPC-C New-Order performance on a 200 server cluster.

level correctness, or consistency.¹ In canonical banking application examples, concurrent, coordination-free withdrawal operations can result in undesirable and “inconsistent” outcomes like negative account balances—application-level anomalies that the database should prevent. To ensure correct behavior, a database system must coordinate the execution of these operations that, if otherwise executed concurrently, could result in inconsistent application state.

This tension between coordination and correctness is evidenced by the range of database concurrency control policies. In traditional database systems, serializable isolation provides concurrent operations (transactions) with the illusion of executing in some serial order [15]. As long as individual transactions maintain correct application state, serializability guarantees correctness [30]. However, each pair of concurrent operations (at least one of which is a write) can potentially compromise serializability and therefore will require coordination to execute [9, 21]. By isolating users at the level of reads and writes, serializability can be overly conservative and may in turn coordinate more than is strictly necessary for consistency [29, 39, 53, 58]. For example, hundreds of users can safely and simultaneously retweet Barack Obama on Twitter without observing a serial ordering of updates to the retweet counter. In contrast, a range of widely-deployed weaker models require less coordination to execute but surface read and write behavior that may in turn compromise consistency [2, 9, 22, 48]. With these alternative models, it is up to users to decide when weakened guarantees are

Can we avoid coordination?

But this is for transactional database systems.

Can we generalize this further?

Coordination Avoidance in Database Systems

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

ABSTRACT

Minimizing coordination, or blocking communication between concurrently executing operations, is key to maximizing scalability, availability, and high performance in database systems. However, uninhibited coordination-free execution can compromise application correctness, or consistency. When is coordination necessary for correctness? The classic use of serializable transactions is sufficient to maintain correctness but is not necessary for all applications, sacrificing potential scalability. In this paper, we develop a formal framework, invariant confluence, that determines whether an application requires coordination for correct execution. By operating on application-level invariants over database states (e.g., integrity constraints), invariant confluence analysis provides a necessary and sufficient condition for safe, coordination-free execution. When programmers specify their application invariants, this analysis allows databases to coordinate only when anomalies that might violate invariants are possible. We analyze the invariant confluence of common invariants and operations from real-world database systems (i.e., integrity constraints) and applications and show that many are invariant confluent and therefore achievable without coordination. We apply these results to a proof-of-concept coordination-avoiding database prototype and demonstrate sizable performance gains compared to serializable execution, notably a 25-fold improvement over prior TPC-C New-Order performance on a 200 server cluster.

level correctness, or consistency.¹ In canonical banking application examples, concurrent, coordination-free withdrawal operations can result in undesirable and “inconsistent” outcomes like negative account balances—application-level anomalies that the database should prevent. To ensure correct behavior, a database system must coordinate the execution of these operations that, if otherwise executed concurrently, could result in inconsistent application state.

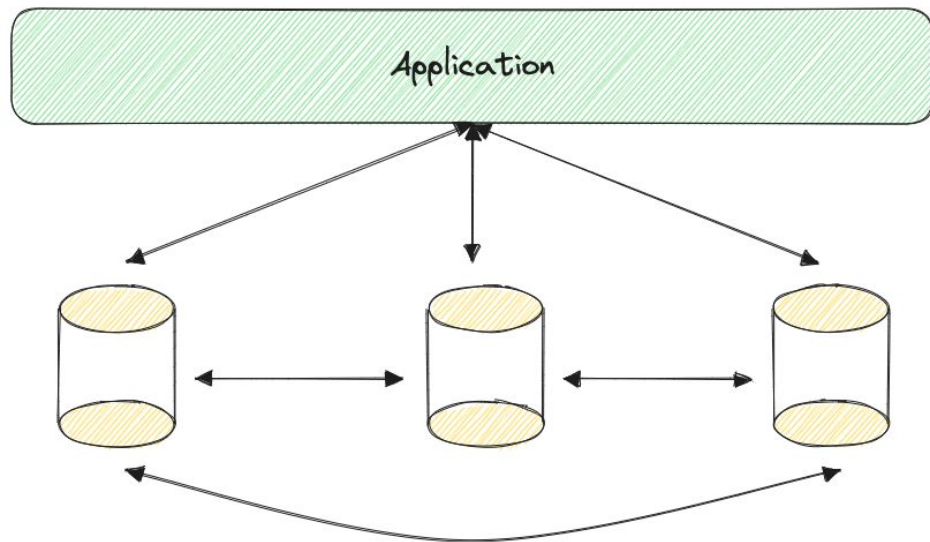
This tension between coordination and correctness is evidenced by the range of database concurrency control policies. In traditional database systems, serializable isolation provides concurrent operations (transactions) with the illusion of executing in some serial order [15]. As long as individual transactions maintain correct application state, serializability guarantees correctness [30]. However, each pair of concurrent operations (at least one of which is a write) can potentially compromise serializability and therefore will require coordination to execute [9, 21]. By isolating users at the level of reads and writes, serializability can be overly conservative and may in turn coordinate more than is strictly necessary for consistency [29, 39, 53, 58]. For example, hundreds of users can safely and simultaneously retweet Barack Obama on Twitter without observing a serial ordering of updates to the retweet counter. In contrast, a range of widely-deployed weaker models require less coordination to execute but surface read and write behavior that may in turn compromise consistency [2, 9, 22, 48]. With these alternative models, it is up to users to decide when weakened guarantees are

Can we avoid coordination?

- Ultimately, we coordinate to achieve memory consistency.

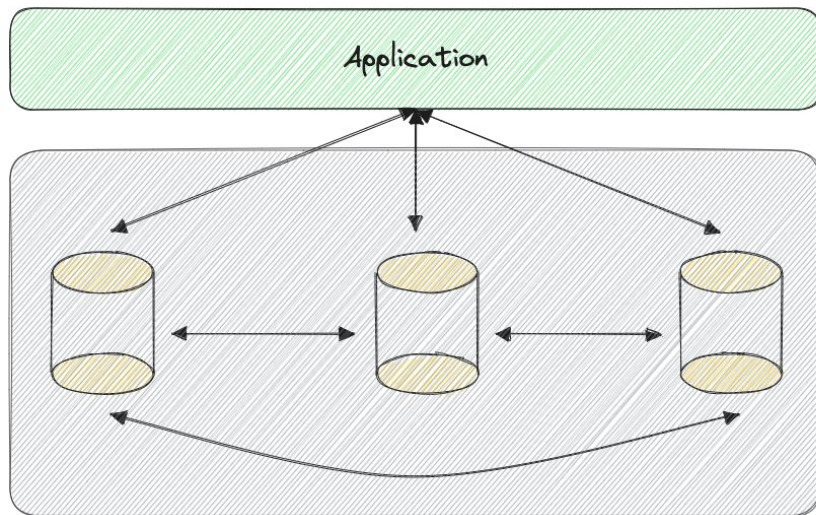
Can we avoid coordination?

- Ultimately, we coordinate to achieve memory consistency.



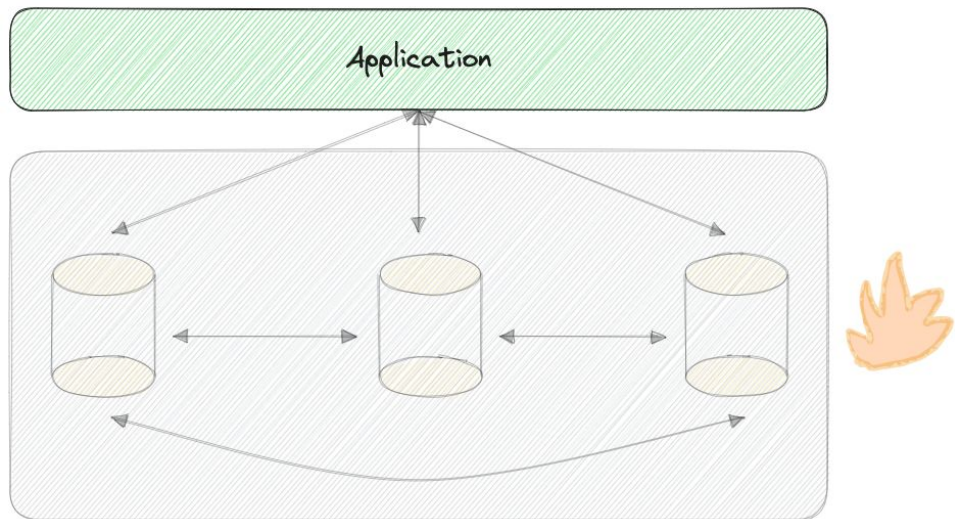
Can we avoid coordination?

- Ultimately, we coordinate to achieve memory consistency.
- And its memory consistency that stands the risk of being violated by all the non-determinism we spoke about.



Can we avoid coordination?

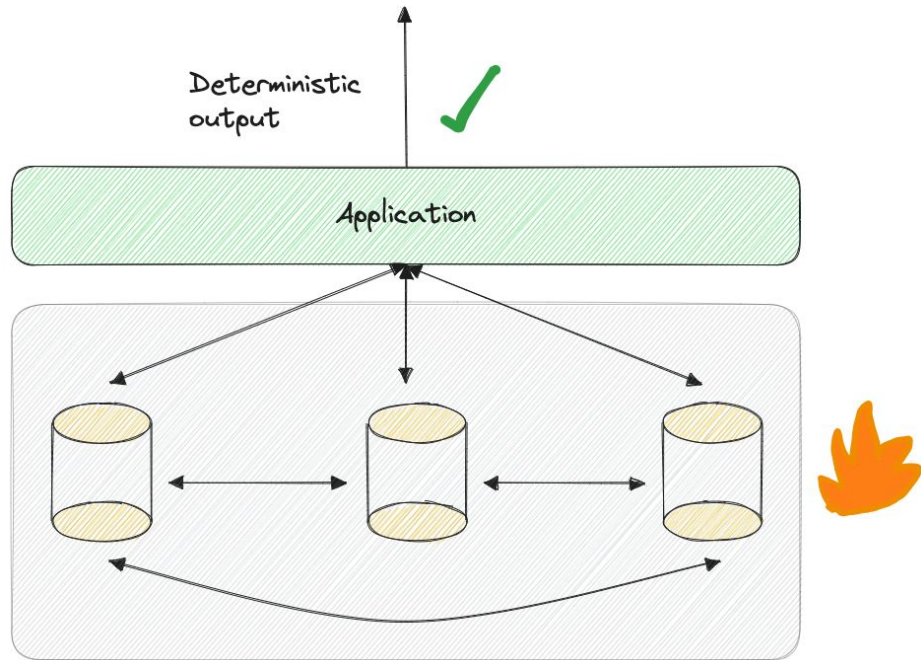
- But what if we move our focus from memory consistency to something called *application-level consistency*?



Can we avoid coordination?

- But what if we move our focus from memory consistency to something called *application-level consistency*?
- Can my program produce deterministic outputs despite non-determinism in the underlying distributed runtime?

Program Confluence.



Can we avoid coordination?

- But what if we move our focus from memory consistency to something called *application-level consistency*?
- Can my program produce deterministic outputs despite non-determinism in the underlying distributed runtime?

Program Confluence.

Keeping CALM: When Distributed Consistency is Easy

Joseph M. Hellerstein
hellerstein@berkeley.edu
UC Berkeley

Peter Alvaro
palvaro@cs.ucsc.edu
UC Santa Cruz

1 INTRODUCTION

Nearly all of the software we use today is part of a distributed system. Apps on your phone participate with hosted services in the cloud; together they form a distributed system. Hosted services themselves are massively distributed systems, often running on machines spread across the globe. "Big data" systems and enterprise databases are distributed across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

Distributed systems are tricky, so their ubiquity should worry us. Multiple unreliable machines are running in parallel, sending messages to each other across network links with arbitrary delays. How can we be confident that our programs do what we want despite this chaos?

This problem is urgent, but it is not new. The traditional answer has been to reduce this complexity with *memory consistency* guarantees: assurances that the accesses to memory (heap variables, database keys, etc) occur in a controlled fashion. However,

simple systems with narrow APIs. Can we avoid coordination more generally, as Hamilton recommends? When?

Surprisingly, this was an open question in distributed systems until relatively recently, due to a narrow focus on storage semantics. We can do better by moving up the stack, setting aside incidental storage details and considering program semantics more holistically. Before we delve into details, we begin with intuition on what is desirable and what is possible.

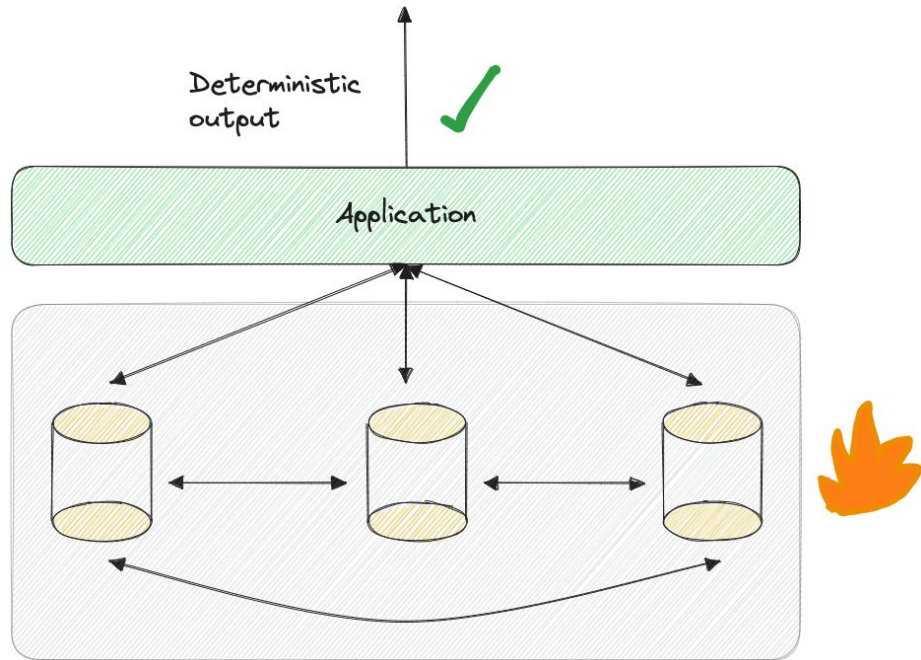
1.2 Stay in Your Lane: The Perfect Freeway

As an analogy, consider driving on a highway during rush hour. If each car would drive forward independently in its lane at the speed limit, everything would be fine: the capacity of the highway could be fully exploited. Unfortunately, there always seem to be drivers who have other places to go than forward! To prevent two cars from being in the same place at the same time, we drivers engage in various forms of coordination when entering traffic, changing lanes, coming to intersections, etc. We adhere to formal protocols, including traffic lights and stop signs. We also frequently engage in ad hoc forms of coordination with neighboring cars by using turn

[cs.DC] 26 Jan 2019

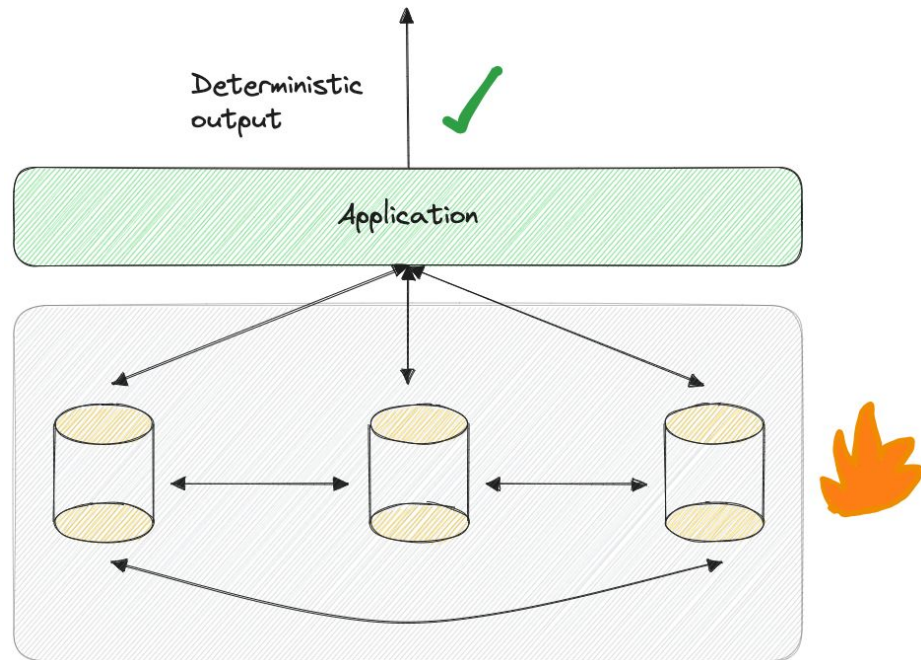
Can we avoid coordination?

Program confluence is pretty cool, but can we define a *class* of programs that are program confluent? A mental framework?



Can we avoid coordination?

Let's take a few examples!



Can we avoid coordination?

Clarification

- “Avoiding coordination” does not mean machines never talk to each other at all.
- Machines communicate periodically – kind of like gossip.
 - More on the frequency of communication later.
- It’s just that for each request, a blocking, potentially sequential, throughput reducing operation is not done.
- Avoiding coordination == can we safely execute a request/query without it being blocking, sequential, throughput reducing?

Can we avoid coordination?

Example 1 – Distributed Deadlock Detection

Can we avoid coordination?

Example 1 – Distributed Deadlock Detection

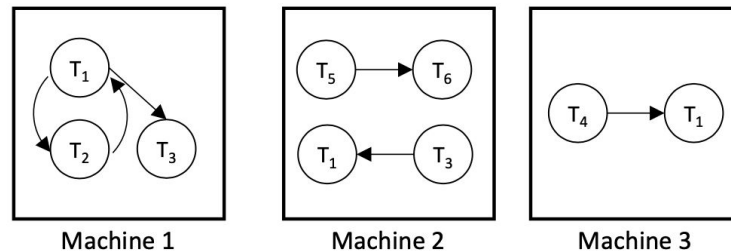


Figure 1: A distributed waits-for graph with replicated nodes and partitioned edges. There are two cycles here: one local to Machine 1 ($\{T_1, T_2\}$), and one that spans Machines 1 and 2 ($\{T_1, T_3\}$).

From Keeping CALM: When Distributed Consistency Is Easy

Can we avoid coordination?

Example 1 – Distributed Deadlock Detection

- Goal is to detect “waits-for” cycles, cycles that can span multiple machines.
- Each machine has a subset of edges in a global waits-for graph.
- Information is accumulated by machines sharing edges with each other.
- Eventually, all machines will have a consistent view of the global waits-for graph.

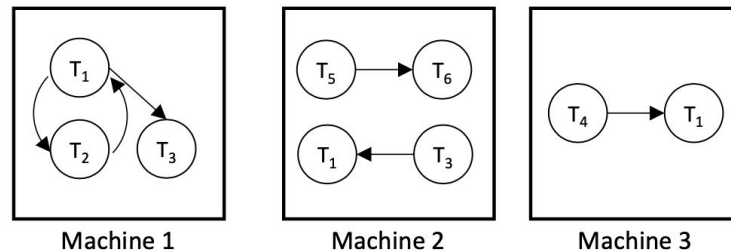


Figure 1: A distributed waits-for graph with replicated nodes and partitioned edges. There are two cycles here: one local to Machine 1 ($\{T_1, T_2\}$), and one that spans Machines 1 and 2 ($\{T_1, T_3\}$).

From *Keeping CALM: When Distributed Consistency Is Easy*

Can we avoid coordination?

Example 1 – Distributed Deadlock Detection

- However, at any point of time, based on the information a machine has accumulated so far, cycles can emerge even without knowing the global view of the graph.
- As and when these cycles emerge, can a local deadlock detector confidently declare that a deadlock has occurred?

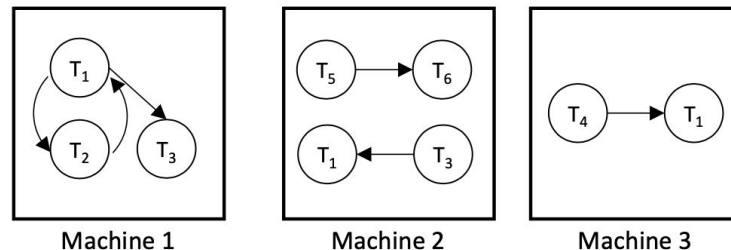


Figure 1: A distributed waits-for graph with replicated nodes and partitioned edges. There are two cycles here: one local to Machine 1 ($\{T_1, T_2\}$), and one that spans Machines 1 and 2 ($\{T_1, T_3\}$).

From *Keeping CALM: When Distributed Consistency Is Easy*

Can we avoid coordination?

Example 1 – Distributed Deadlock Detection

- Turns out it can! But what about race conditions? What if information that we don't yet know, change our decision of having detected a deadlock? Do I need to coordinate with other nodes before declaring a deadlock?

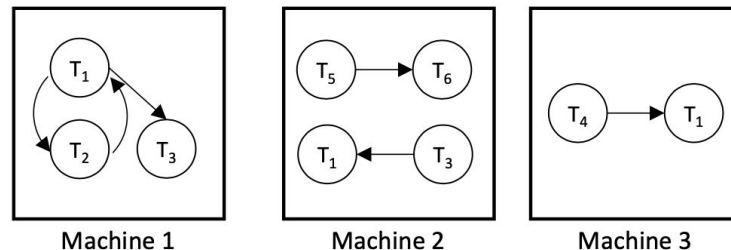


Figure 1: A distributed waits-for graph with replicated nodes and partitioned edges. There are two cycles here: one local to Machine 1 ($\{T_1, T_2\}$), and one that spans Machines 1 and 2 ($\{T_1, T_3\}$).

From *Keeping CALM: When Distributed Consistency Is Easy*

Can we avoid coordination?

Example 1 – Distributed Deadlock Detection

- Turns out it can! But what about race conditions? What if information that we don't yet know, change our decision of having detected a deadlock? Do I need to coordinate with other nodes before declaring a deadlock?
- No need to coordinate. Any decision declared based on partial/local state is still valid. Partial information in this case is always an under-approximation of the global state.

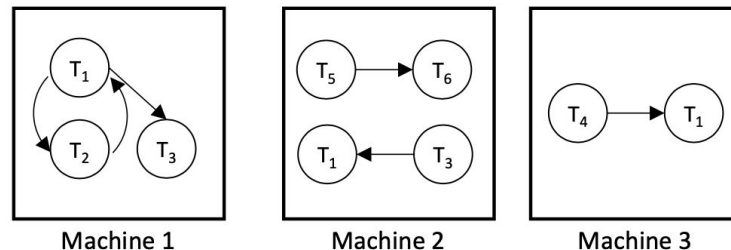


Figure 1: A distributed waits-for graph with replicated nodes and partitioned edges. There are two cycles here: one local to Machine 1 ($\{T_1, T_2\}$), and one that spans Machines 1 and 2 ($\{T_1, T_3\}$).

From *Keeping CALM: When Distributed Consistency Is Easy*

Can we avoid coordination?

Example 2 – Distributed Garbage Collection

Can we avoid coordination?

Example 2 – Distributed Garbage Collection

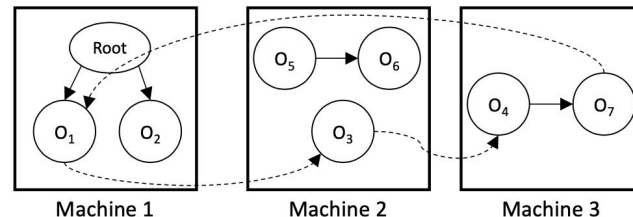


Figure 2: A distributed object reference graph with remote references (dotted arrows). The fact that object O_3 is reachable from Root can be established without any information from Machine 3. Objects O_5 and O_6 are garbage, which can only be established by knowing the entire graph.

From Keeping CALM: When Distributed Consistency Is Easy

Can we avoid coordination?

Example 2 – Distributed Garbage Collection

- Goal is to detect objects that are disconnected from “root”.
- Again, references to objects can span multiple machines.
- A machine’s local view contains only a subset of edges of the global reference graph.
- As before, machines exchange their local copies of edges to accumulate information.
- Eventually, all machines will have a consistent view of the global reference graph.

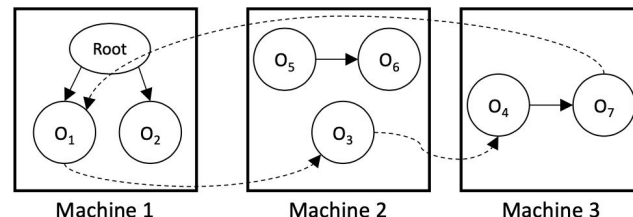


Figure 2: A distributed object reference graph with remote references (dotted arrows). The fact that object O_3 is reachable from Root can be established without any information from Machine 3. Objects O_5 and O_6 are garbage, which can only be established by knowing the entire graph.

From Keeping CALM: When Distributed Consistency Is Easy

Can we avoid coordination?

Example 2 – Distributed Garbage Collection

- However, if at any point, a machine detects that a local object is disconnected from the root, can it declare that this is garbage and deallocate it?
- Can a local garbage collector make decisions to deallocate local objects without complete view of the global reference graph? Can we avoid coordination?
- What about race conditions? Can information we don't yet know cause us to change our mind?

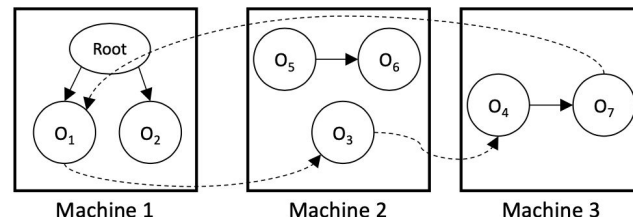


Figure 2: A distributed object reference graph with remote references (dotted arrows). The fact that object O_3 is reachable from Root can be established without any information from Machine 3. Objects O_5 and O_6 are garbage, which can only be established by knowing the entire graph.

From Keeping CALM: When Distributed Consistency Is Easy

Can we avoid coordination?

Example 2 – Distributed Garbage Collection

- In this case, we need to coordinate!
- The reason for this is that a decision made on incomplete information can be invalidated by arrival of new information.
- The local state is not an under-approximation of the global state.

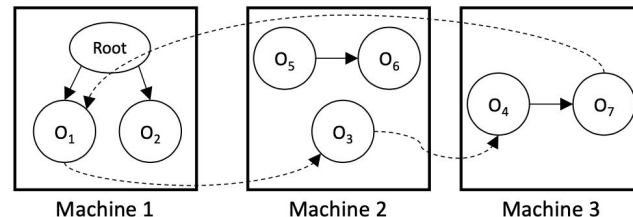


Figure 2: A distributed object reference graph with remote references (dotted arrows). The fact that object O_3 is reachable from Root can be established without any information from Machine 3. Objects O_5 and O_6 are garbage, which can only be established by knowing the entire graph.

From Keeping CALM: When Distributed Consistency Is Easy

Can we avoid coordination?

Question: *What is the family of problems that can be consistently computed in a distributed fashion without coordination, and what problems lie outside that family?*

Keeping CALM: When Distributed Consistency is Easy

Joseph M. Hellerstein
hellerstein@berkeley.edu
UC Berkeley

Peter Alvaro
palvaro@cs.ucsc.edu
UC Santa Cruz

1 INTRODUCTION

Nearly all of the software we use today is part of a distributed system. Apps on your phone participate with hosted services in the cloud; together they form a distributed system. Hosted services themselves are massively distributed systems, often running on machines spread across the globe. "Big data" systems and enterprise databases are distributed across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

Distributed systems are tricky, so their ubiquity should worry us. Multiple unreliable machines are running in parallel, sending messages to each other across network links with arbitrary delays. How can we be confident that our programs do what we want despite this chaos?

This problem is urgent, but it is not new. The traditional answer has been to reduce this complexity with *memory consistency* guarantees: assurances that the accesses to memory (heap variables, database keys, etc) occur in a controlled fashion. However,

simple systems with narrow APIs. Can we avoid coordination more generally, as Hamilton recommends? When?

Surprisingly, this was an open question in distributed systems until relatively recently, due to a narrow focus on storage semantics. We can do better by moving up the stack, setting aside incidental storage details and considering program semantics more holistically. Before we delve into details, we begin with intuition on what is desirable and what is possible.

1.2 Stay in Your Lane: The Perfect Freeway

As an analogy, consider driving on a highway during rush hour. If each car would drive forward independently in its lane at the speed limit, everything would be fine: the capacity of the highway could be fully exploited. Unfortunately, there always seem to be drivers who have other places to go than forward! To prevent two cars from being in the same place at the same time, we drivers engage in various forms of coordination when entering traffic, changing lanes, coming to intersections, etc. We adhere to formal protocols, including traffic lights and stop signs. We also frequently engage in ad hoc forms of coordination with neighboring cars by using turn

[cs.DC] 26 Jan 2019

Can we avoid coordination?

Theorem 1: *Consistency As Logical Monotonicity (CALM). A program has a consistent, coordination-free distributed implementation if and only if it is monotonic.*

Keeping CALM: When Distributed Consistency is Easy

Joseph M. Hellerstein
hellerstein@berkeley.edu
UC Berkeley

Peter Alvaro
palvaro@cs.ucsc.edu
UC Santa Cruz

1 INTRODUCTION

Nearly all of the software we use today is part of a distributed system. Apps on your phone participate with hosted services in the cloud; together they form a distributed system. Hosted services themselves are massively distributed systems, often running on machines spread across the globe. "Big data" systems and enterprise databases are distributed across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

Distributed systems are tricky, so their ubiquity should worry us. Multiple unreliable machines are running in parallel, sending messages to each other across network links with arbitrary delays. How can we be confident that our programs do what we want despite this chaos?

This problem is urgent, but it is not new. The traditional answer has been to reduce this complexity with *memory consistency* guarantees: assurances that the accesses to memory (heap variables, database keys, etc) occur in a controlled fashion. However,

simple systems with narrow APIs. Can we avoid coordination more generally, as Hamilton recommends? When?

Surprisingly, this was an open question in distributed systems until relatively recently, due to a narrow focus on storage semantics. We can do better by moving up the stack, setting aside incidental storage details and considering program semantics more holistically. Before we delve into details, we begin with intuition on what is desirable and what is possible.

1.2 Stay in Your Lane: The Perfect Freeway

As an analogy, consider driving on a highway during rush hour. If each car would drive forward independently in its lane at the speed limit, everything would be fine: the capacity of the highway could be fully exploited. Unfortunately, there always seem to be drivers who have other places to go than forward! To prevent two cars from being in the same place at the same time, we drivers engage in various forms of coordination when entering traffic, changing lanes, coming to intersections, etc. We adhere to formal protocols, including traffic lights and stop signs. We also frequently engage in ad hoc forms of coordination with neighboring cars by using turn

[cs.DC] 26 Jan 2019

Can we avoid coordination?

Theorem 1: *Consistency As Logical Monotonicity (CALM)*. A program has a consistent, coordination-free distributed implementation if and only if it is monotonic.

“Reasoners draw conclusions defeasibly when they reserve the right to retract them in the light of further information”

Non-monotonic Logic, Stanford Encyclopedia of Philosophy

[\(https://plato.stanford.edu/entries/logic-nonmonotonic/\)](https://plato.stanford.edu/entries/logic-nonmonotonic/)

Keeping CALM: When Distributed Consistency is Easy

Joseph M. Hellerstein
hellerstein@berkeley.edu
UC Berkeley

Peter Alvaro
palvaro@cs.ucsc.edu
UC Santa Cruz

1 INTRODUCTION

Nearly all of the software we use today is part of a distributed system. Apps on your phone participate with hosted services in the cloud; together they form a distributed system. Hosted services themselves are massively distributed systems, often running on machines spread across the globe. “Big data” systems and enterprise databases are distributed across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

Distributed systems are tricky, so their ubiquity should worry us. Multiple unreliable machines are running in parallel, sending messages to each other across network links with arbitrary delays. How can we be confident that our programs do what we want despite this chaos?

This problem is urgent, but it is not new. The traditional answer has been to reduce this complexity with *memory consistency* guarantees: assurances that the accesses to memory (heap variables, database keys, etc) occur in a controlled fashion. However,

simple systems with narrow APIs. Can we avoid coordination more generally, as Hamilton recommends? When?

Surprisingly, this was an open question in distributed systems until relatively recently, due to a narrow focus on storage semantics. We can do better by moving up the stack, setting aside incidental storage details and considering program semantics more holistically. Before we delve into details, we begin with intuition on what is desirable and what is possible.

1.2 Stay in Your Lane: The Perfect Freeway

As an analogy, consider driving on a highway during rush hour. If each car would drive forward independently in its lane at the speed limit, everything would be fine: the capacity of the highway could be fully exploited. Unfortunately, there always seem to be drivers who have other places to go than forward! To prevent two cars from being in the same place at the same time, we drivers engage in various forms of coordination when entering traffic, changing lanes, coming to intersections, etc. We adhere to formal protocols, including traffic lights and stop signs. We also frequently engage in ad hoc forms of coordination with neighboring cars by using turn

[cs.DC] 26 Jan 2019

Can we avoid coordination?

Theorem 1: *Consistency As Logical Monotonicity (CALM). A program has a consistent, coordination-free distributed implementation if and only if it is monotonic.*

Definition 1: *A program P is monotonic if for any input sets S, T where $S \subseteq T$, $P(S) \subseteq P(T)$.*

Keeping CALM: When Distributed Consistency is Easy

Joseph M. Hellerstein
hellerstein@berkeley.edu
UC Berkeley

Peter Alvaro
palvaro@cs.ucsc.edu
UC Santa Cruz

1 INTRODUCTION

Nearly all of the software we use today is part of a distributed system. Apps on your phone participate with hosted services in the cloud; together they form a distributed system. Hosted services themselves are massively distributed systems, often running on machines spread across the globe. "Big data" systems and enterprise databases are distributed across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

Distributed systems are tricky, so their ubiquity should worry us. Multiple unreliable machines are running in parallel, sending messages to each other across network links with arbitrary delays. How can we be confident that our programs do what we want despite this chaos?

This problem is urgent, but it is not new. The traditional answer has been to reduce this complexity with *memory consistency* guarantees: assurances that the accesses to memory (heap variables, database keys, etc) occur in a controlled fashion. However,

simple systems with narrow APIs. Can we avoid coordination more generally, as Hamilton recommends? When?

Surprisingly, this was an open question in distributed systems until relatively recently, due to a narrow focus on storage semantics. We can do better by moving up the stack, setting aside incidental storage details and considering program semantics more holistically. Before we delve into details, we begin with intuition on what is desirable and what is possible.

1.2 Stay in Your Lane: The Perfect Freeway

As an analogy, consider driving on a highway during rush hour. If each car would drive forward independently in its lane at the speed limit, everything would be fine: the capacity of the highway could be fully exploited. Unfortunately, there always seem to be drivers who have other places to go than forward! To prevent two cars from being in the same place at the same time, we drivers engage in various forms of coordination when entering traffic, changing lanes, coming to intersections, etc. We adhere to formal protocols, including traffic lights and stop signs. We also frequently engage in ad hoc forms of coordination with neighboring cars by using turn

[cs.DC] 26 Jan 2019

Can we avoid coordination?

- Remember – the need to coordinate arises from an intrinsic need to gather missing information.

Can we avoid coordination?

- Remember – the need to coordinate arises from an intrinsic need to gather missing information.
- As a result, monotonic programs are “safe” in the face of missing information and can proceed without coordination.

Can we avoid coordination?

- Remember – the need to coordinate arises from an intrinsic need to gather missing information.
- As a result, monotonic programs are “safe” in the face of missing information and can proceed without coordination.
- Non-monotonic programs on the other hand tend to “change their mind” in the face of new information, they *need* to ensure they know the global state before taking any decisions.

Can we avoid coordination?

- Remember – the need to coordinate arises from an intrinsic need to gather missing information.
- As a result, monotonic programs are “safe” in the face of missing information and can proceed without coordination.
- Non-monotonic programs on the other hand tend to “change their mind” in the face of new information, they *need* to ensure they know the global state before taking any decisions.
- Additionally, because non-monotonicity leads to “change in mind”, they are also sensitive to the order in which inputs are processed – another intrinsic motivator for coordination. Monotonic programs are immune to this as well! They only care about the *content* of inputs, not the order.

Interlude – CRDTs, a primer.

CRDTs

Note: this hopes to be an intuitive introduction to CRDTs, resources for a more concrete and mathematically sound introduction to CRDTs are linked towards the end!

CRDTs

- Conflict Free Replicated Datatypes.
- These are replicated structures that provide guarantees to be eventually consistent without the need for coordination.



Conflict-free Replicated Data Types

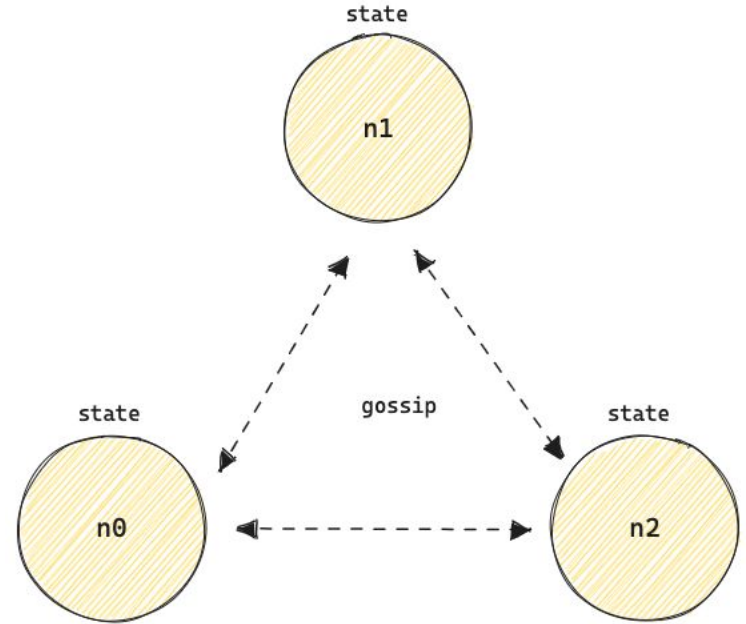
Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski

► To cite this version:

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. Conflict-free Replicated Data Types. SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems, Oct 2011, Grenoble, France. pp.386-400, 10.1007/978-3-642-24550-3_29 . hal-00932836

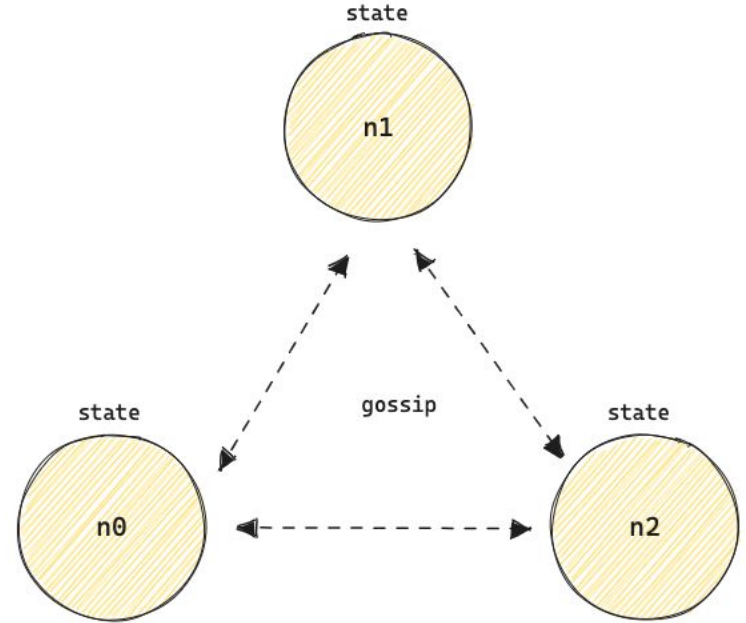
CRDTs

- Conflict Free Replicated Datatypes.
- These are replicated structures that provide guarantees to be eventually consistent without the need for coordination.
- Replicas gossip their state and all become consistent eventually.



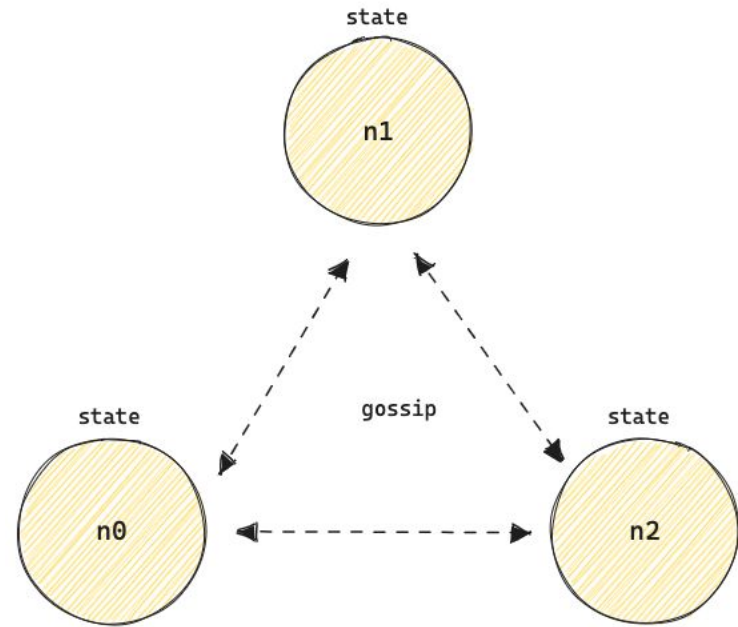
CRDTs

- These are called state-based CRDTs, there's also something called operation-based CRDTs.
- We will only talk about state-based CRDTs today to simplify things.



CRDTs

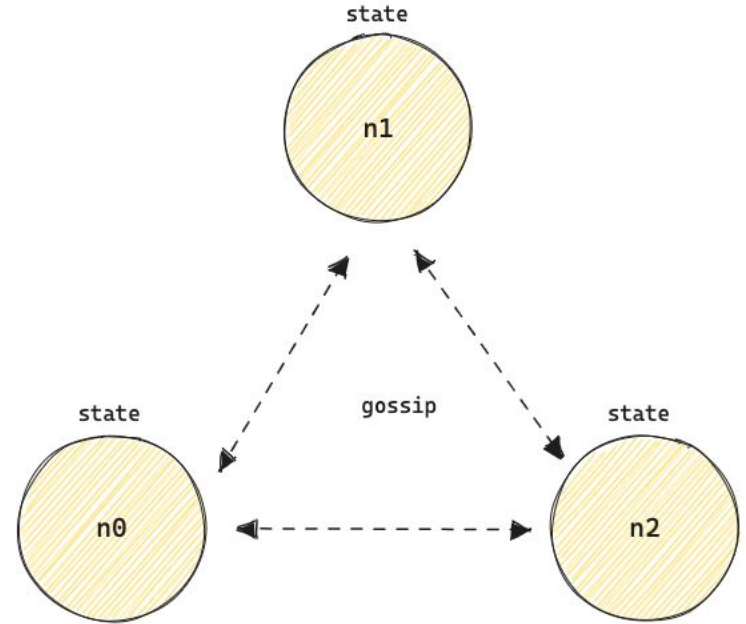
To understand CRDTs, let's understand how its API is defined:



CRDTs

To understand CRDTs, let's understand how its API is defined:

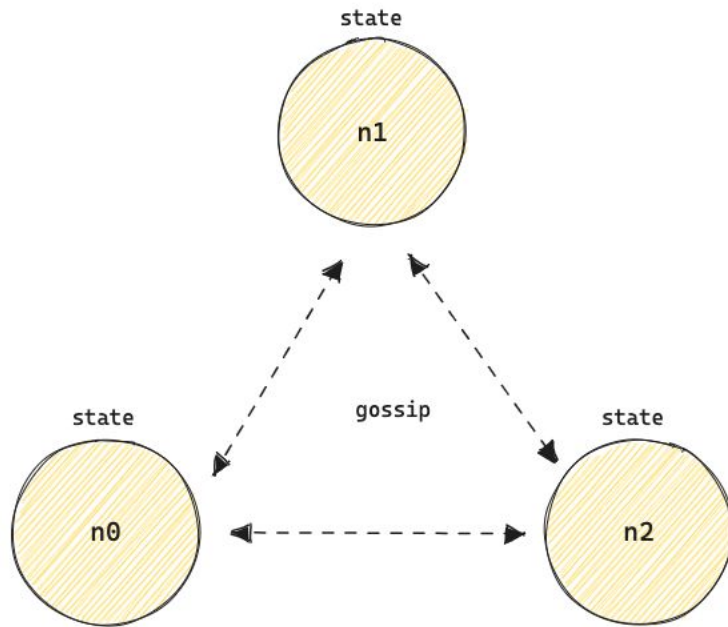
- Each function is executed locally.



CRDTs

To understand CRDTs, let's understand how its API is defined:

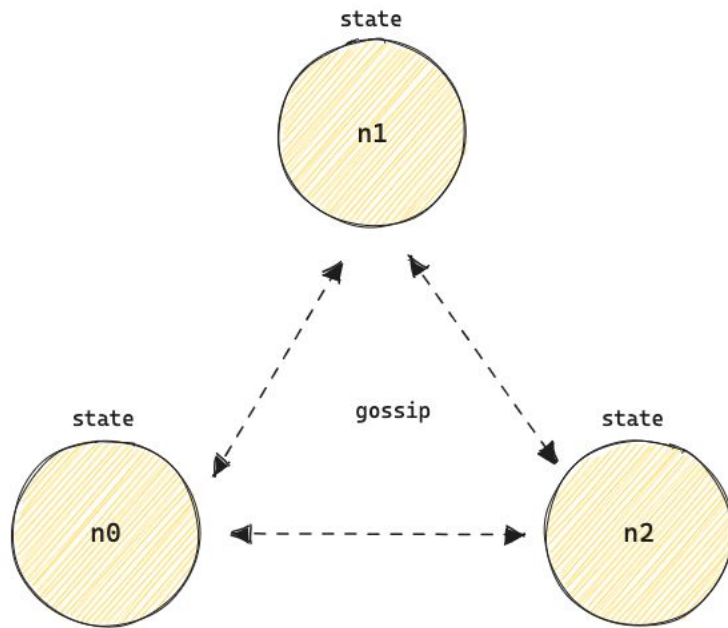
- Each function is executed locally.
- **op**: Clients use this to modify the state of the CRDT. Must be monotonic.



CRDTs

To understand CRDTs, let's understand how its API is defined:

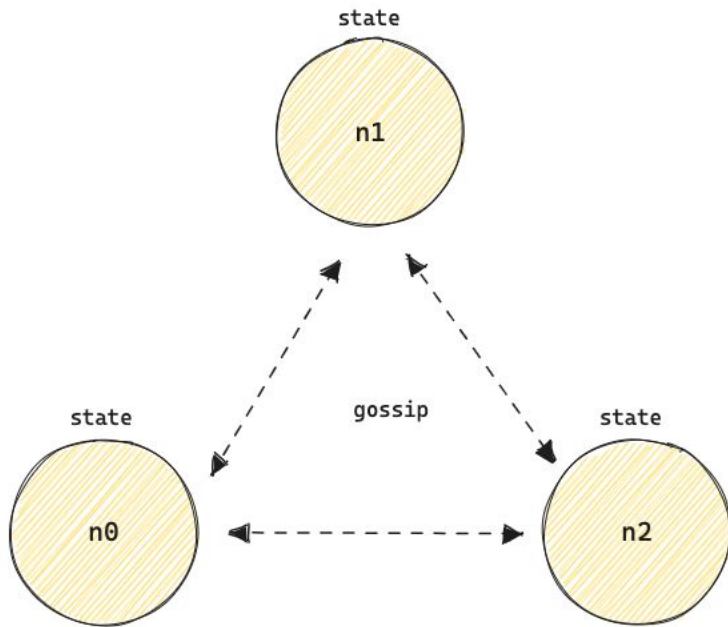
- Each function is executed locally.
- **op**: Clients use this to modify the state of the CRDT. Must be monotonic.
- **query**: Does not modify state, only returns some result that might depend on state.



CRDTs

To understand CRDTs, let's understand how its API is defined:

- Each function is executed locally.
- **op**: Clients use this to modify the state of the CRDT. Must be monotonic.
- **query**: Does not modify state, only returns some result that might depend on state.
- **merge**: Takes a value, merges it with existing state and produces new state. Must be Associative, Commutative and Idempotent (ACI).



CRDTs

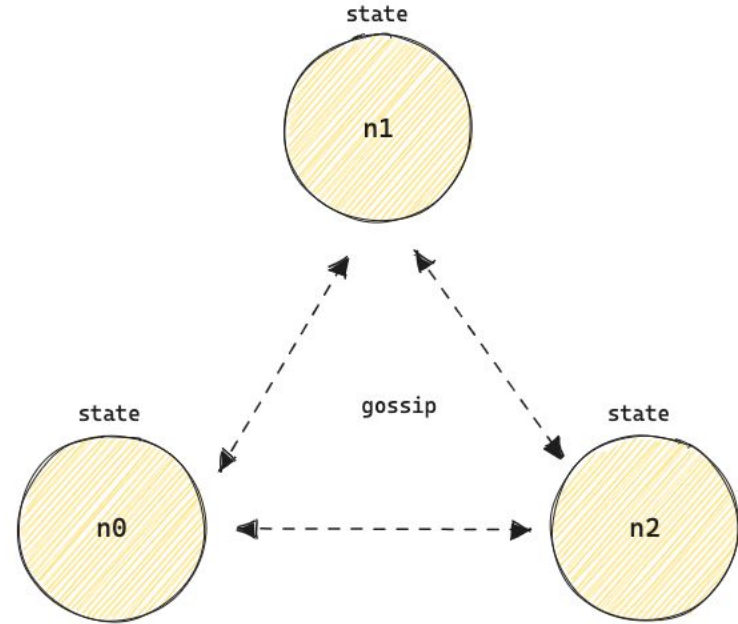
merge: Takes a value, merges it with existing state and produces new state. Must be Associative, Commutative and Idempotent (ACI).

If $\&$ is the merge function and a, b, c are updates to the CRDT state:

Associative: $a \& (b \& c) = (a \& b) \& c$

Commutative: $a \& b = b \& a$

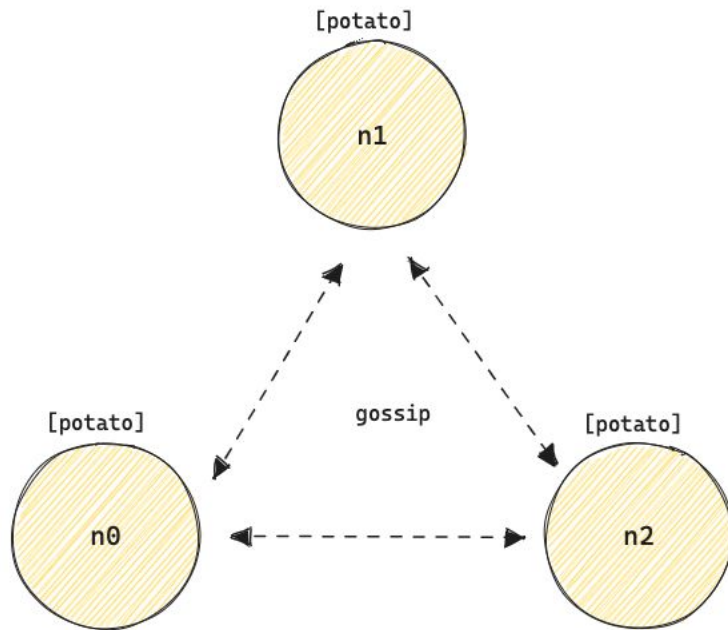
Idempotent: $a \& a = a$



CRDTs

CRDT example: a grow-only replicated set of values

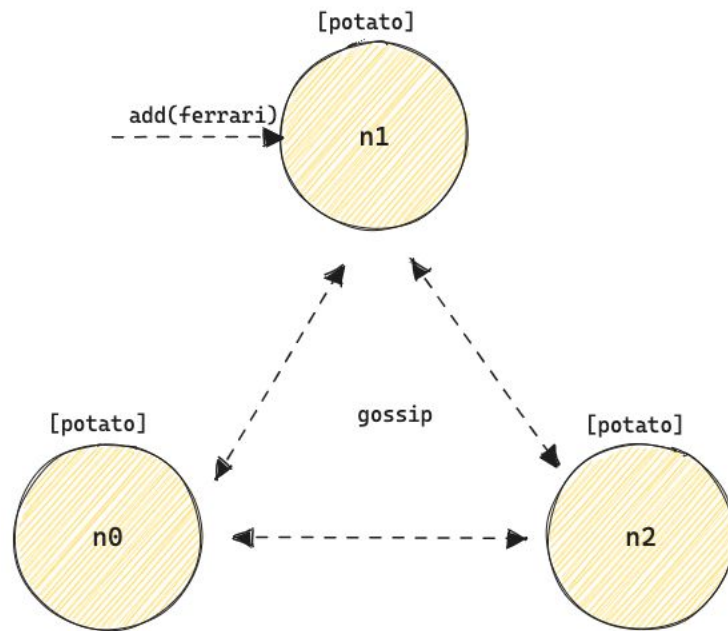
- We have a shopping cart that (for now) users can only add values to.
- The contents of this shopping cart are replicated for latency and availability purposes.



CRDTs

CRDT example: a grow-only replicated set of values

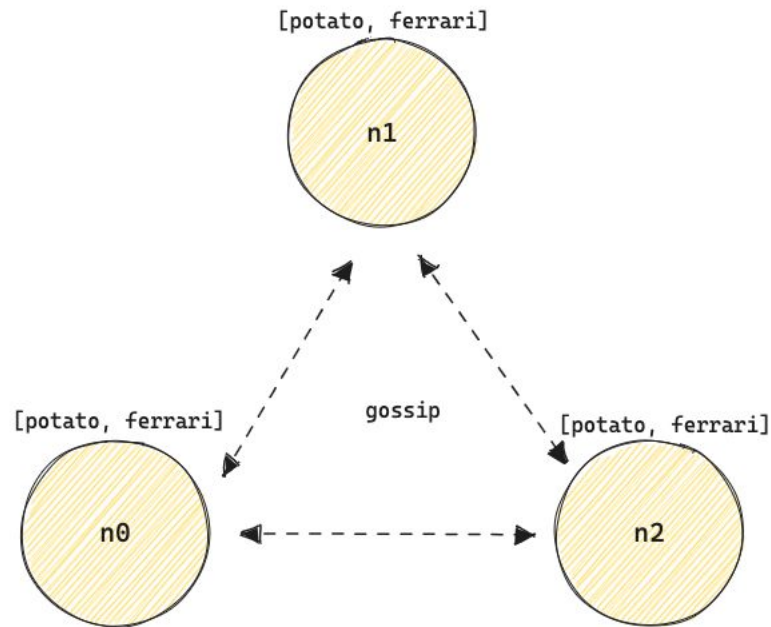
- We have a shopping cart that (for now) users can only add values to.
- The contents of this shopping cart are replicated for latency and availability purposes.



CRDTs

CRDT example: a grow-only replicated set of values

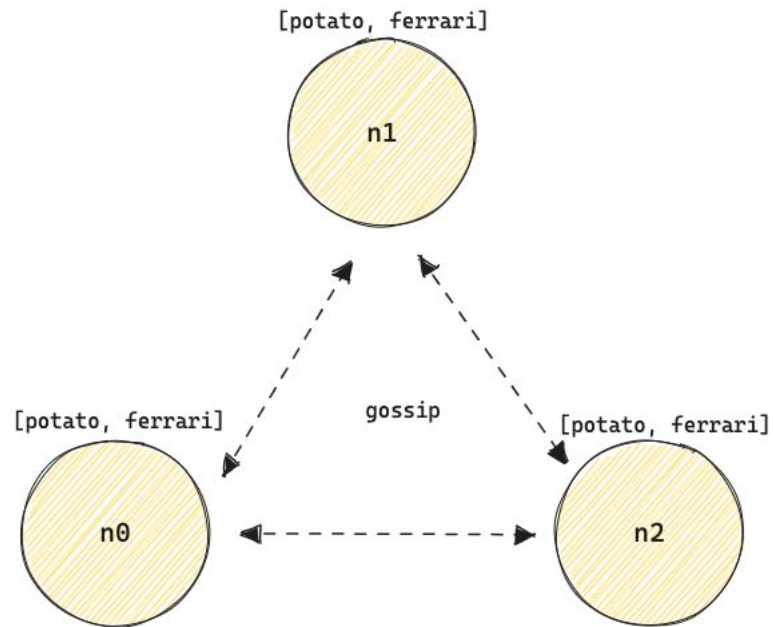
- We have a shopping cart that (for now) users can only add values to.
- The contents of this shopping cart are replicated for latency and availability purposes.



CRDTs

API for this CRDT:

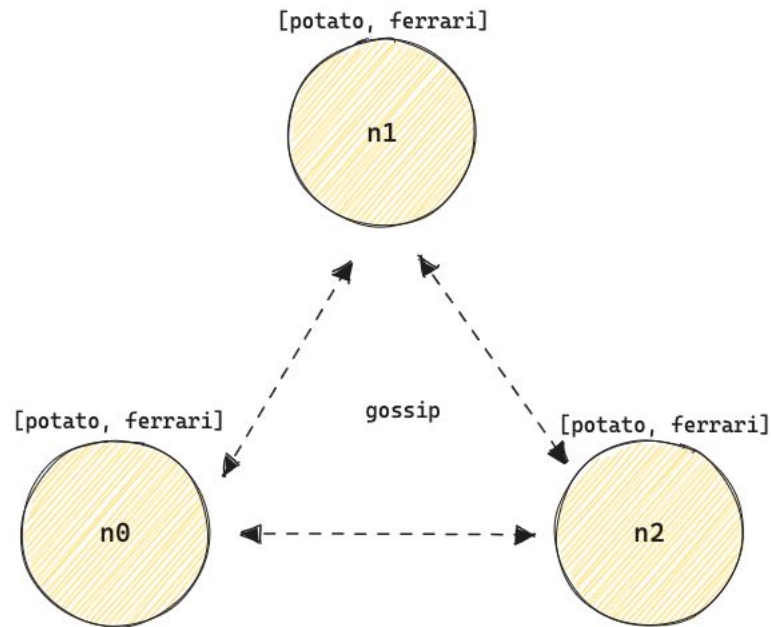
- **op**: `add(item T) { adds.insert(item) }`



CRDTs

API for this CRDT:

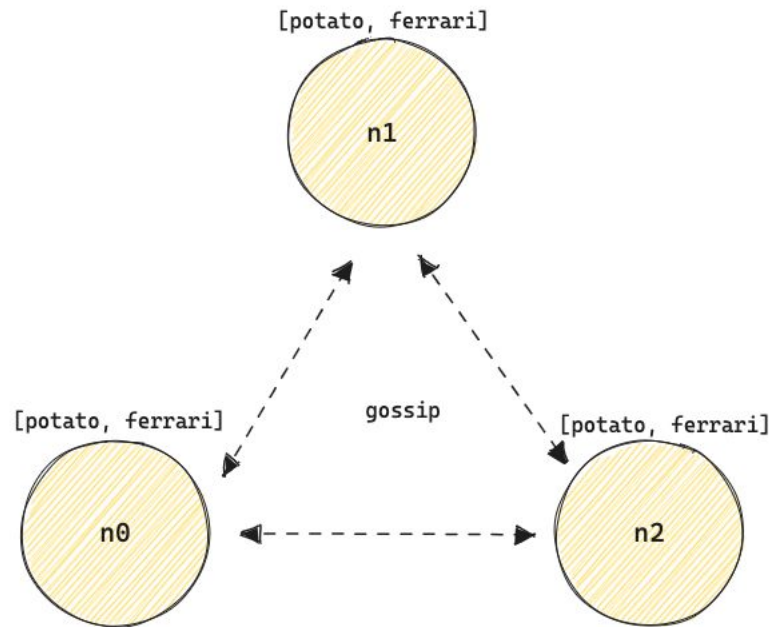
- **op:** `add(item T) { adds.insert(item) }`
- **query:** `read() []T { return adds }`



CRDTs

API for this CRDT:

- **op:** `add(item T) { adds.insert(item) }`
- **query:** `read() []T { return adds }`
- **merge:** `union(item) { adds.union(item) }`



CRDTs

API for this CRDT:

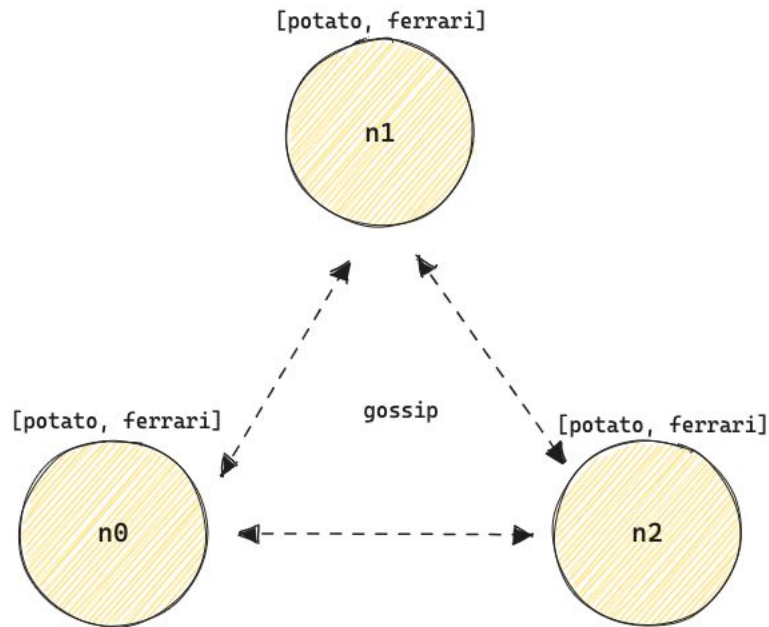
merge: `union(item)`

If $\& = \text{union}$ is the merge function and x, y, z are additions to the set:

Associative: $x \& (y \& z) = (x \& y) \& z$
 $= \{x, y, z\}$

Commutative: $x \& y = y \& x = \{x, y\}$

Idempotent: $x \& x = \{x\}$



CRDTs

Mathematically, you can represent this CRDT as:

$(\{x, y, z\}, \&)$

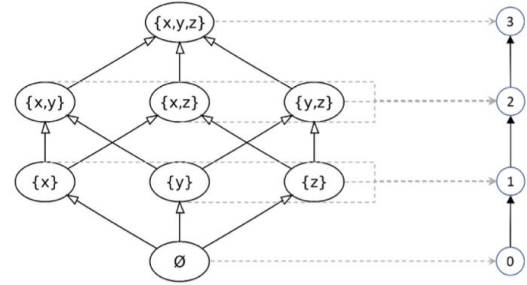


Figure 2: Hasse diagrams for G-Set and a cardinality counter, and a monotone function between them (dashed lines).

CRDTs

Mathematically, you can represent this CRDT as:

$(\{x, y, z\}, \&)$

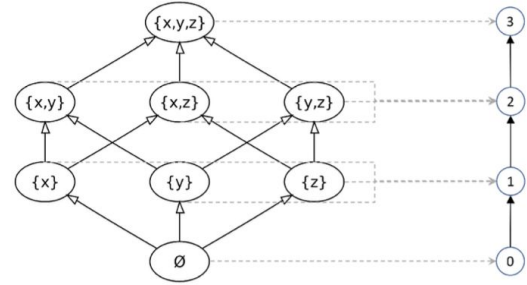


Figure 2: Hasse diagrams for G-Set and a cardinality counter, and a monotone function between them (dashed lines).

The only way is up.

Popularity of CRDTs

Popularity of CRDTs

- Used as building blocks by distributed systems developers: Akka, Dynamo, Redis.
- Used by industry – PayPal, League of Legends, FlightTracker (inside Meta).
- Used in collaborative document editing.

Why The Popularity of CRDTs?

Why The Popularity of CRDTs?

- An easy to explain API.
- A promise of formal safety guarantees (eventual convergence) – its attractive to latch onto “*guaranteed to converge, all replicas eventually consistent*”
- Helps deal with non-determinism that comes with eventually consistent systems: re-ordering, duplication, late-arriving updates – ACI merge function handles that!

A Few Gotchas

A Few Gotchas

“guaranteed to converge, all replicas eventually consistent”

- Because CRDTs have become so popular, it starts becoming simpler to misread what the actual guarantees provided by CRDTs are.

A Few Gotchas

“guaranteed to converge, all replicas eventually consistent”

- Because CRDTs have become so popular, it starts becoming simpler to misread what the actual guarantees provided by CRDTs are.
- This is a *storage* guarantee. This is not a guarantee that is provided to readers of the state of CRDTs.

A Few Gotchas

So as a developer, if I wanted to have such guarantees for reading state as well:

A Few Gotchas

So as a developer, if I wanted to have such guarantees for reading state as well:

- I understand the system is eventual, I've accepted stale reads.

A Few Gotchas

So as a developer, if I wanted to have such guarantees for reading state as well:

- I understand the system is eventual, I've accepted stale reads.
- However, if I'm getting the guarantee of no coordination, I expect my reads to never go back in time, otherwise I'll have to coordinate.

A Few Gotchas

So as a developer, if I wanted to have such guarantees for reading state as well:

- I understand the system is eventual, I've accepted stale reads.
- However, if I'm getting the guarantee of no coordination, I expect my reads to never go back in time, otherwise I'll have to coordinate.
- Because I am not coordinating, I also expect no anomalies in my state – all conflicts are handled. The system is basically equivalent to some sequential execution.

A Few Gotchas

Wait...

Am I expecting sequential consistency?

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

Index Terms—Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called *sequentially consistent*. The sequentiality

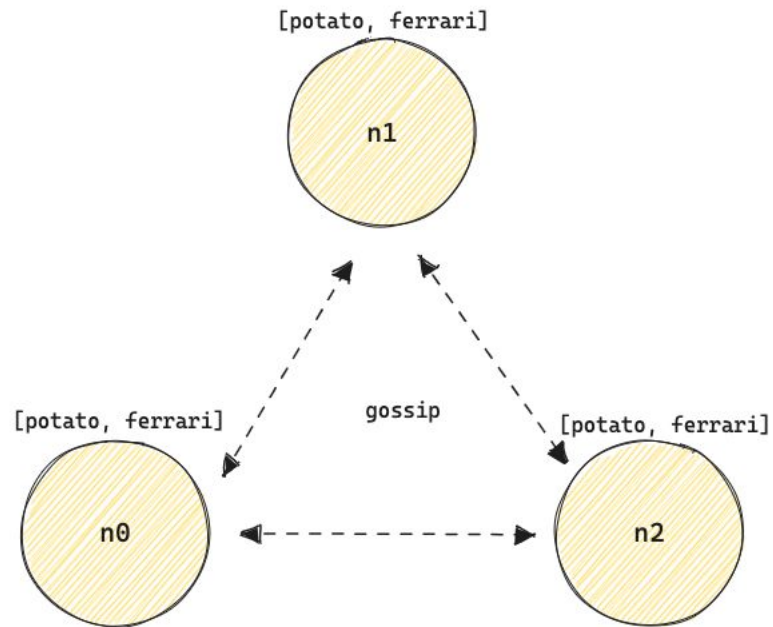
Manuscript received September 28, 1977; revised May 8, 1979.

The author is with the Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

A Few Gotchas

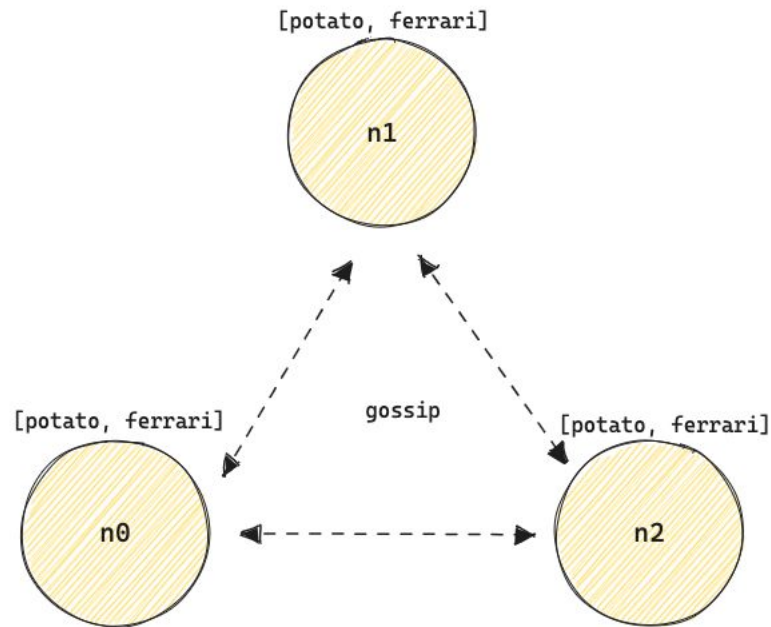
However, CRDTs by themselves provide no such guarantees to *readers*.

A Few Gotchas



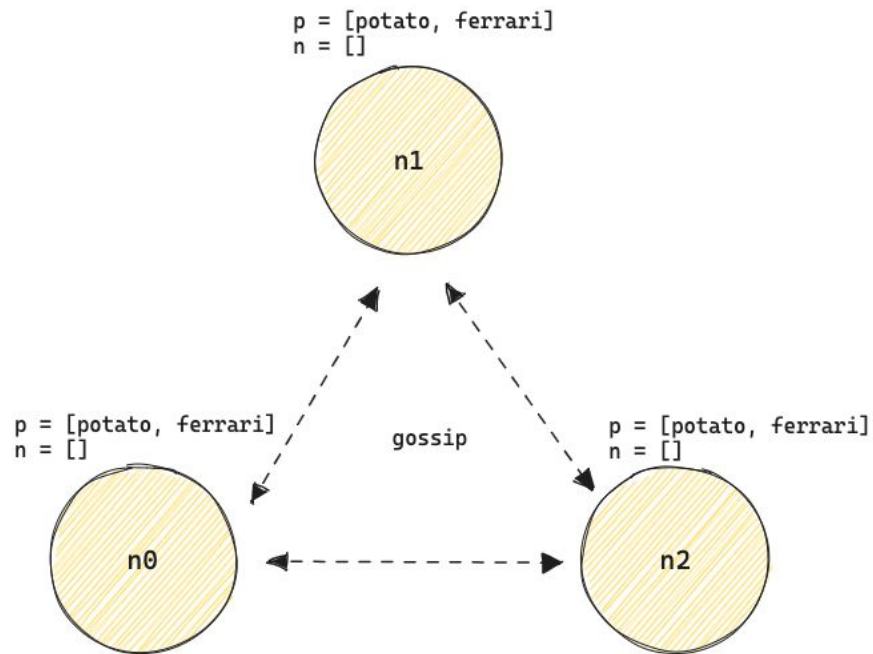
A Few Gotchas

- What if I decide that a Ferrari is probably not the best purchase and I want to remove it from my cart?
- Deletions from a set violate monotonicity, we are going back on our state of world!



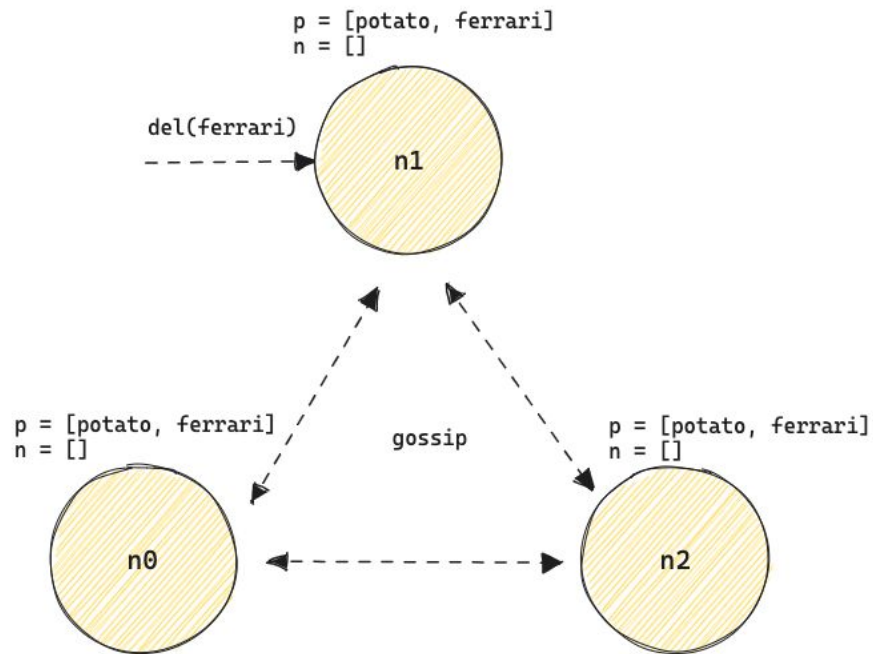
A Few Gotchas

- What if I decide that a Ferrari is probably not the best purchase and I want to remove it from my cart?
- Deletions from a set violate monotonicity, we are going back on our state of world!
- Another grow-only set but for deletions?



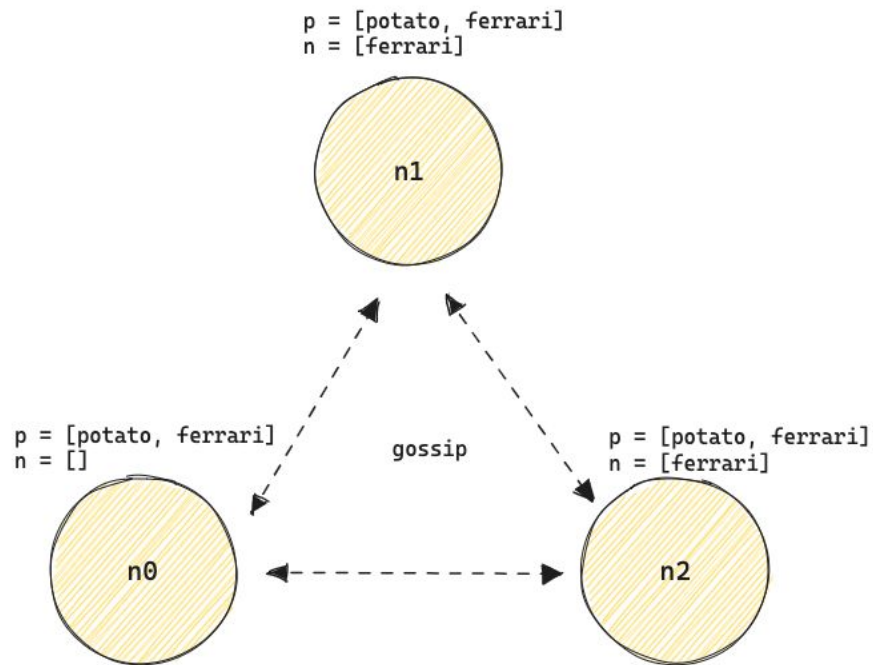
A Few Gotchas

- What if I decide that a Ferrari is probably not the best purchase and I want to remove it from my cart?
- Deletions from a set violate monotonicity, we are going back on our state of world!
- Another grow-only set but for deletions?



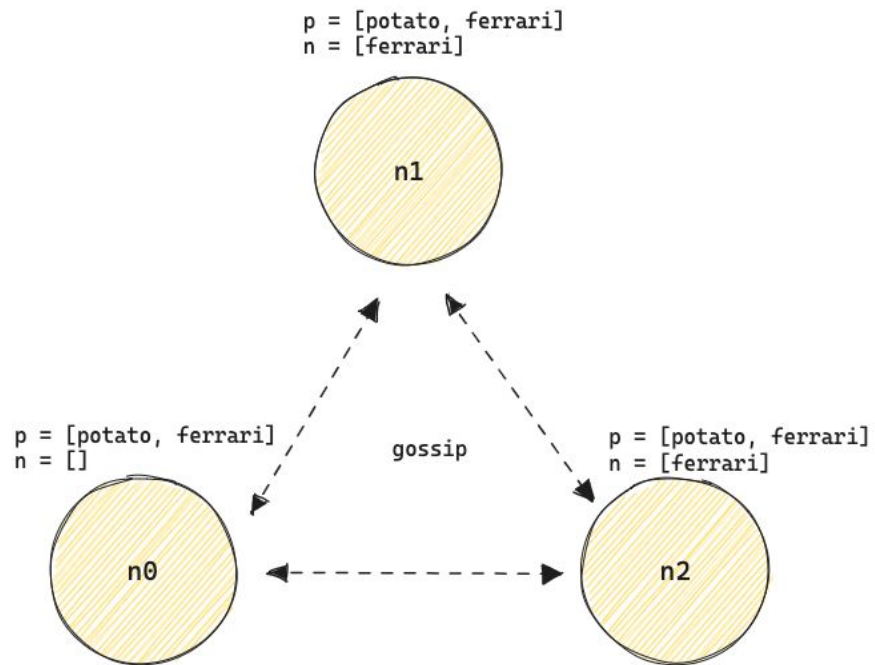
A Few Gotchas

- What if I decide that a Ferrari is probably not the best purchase and I want to remove it from my cart?
- Deletions from a set violate monotonicity, we are going back on our state of world!
- Another grow-only set but for deletions?



A Few Gotchas

API for this CRDT:



A Few Gotchas

API for this CRDT:

op:

```
add(item T) { adds.insert(item) }
```

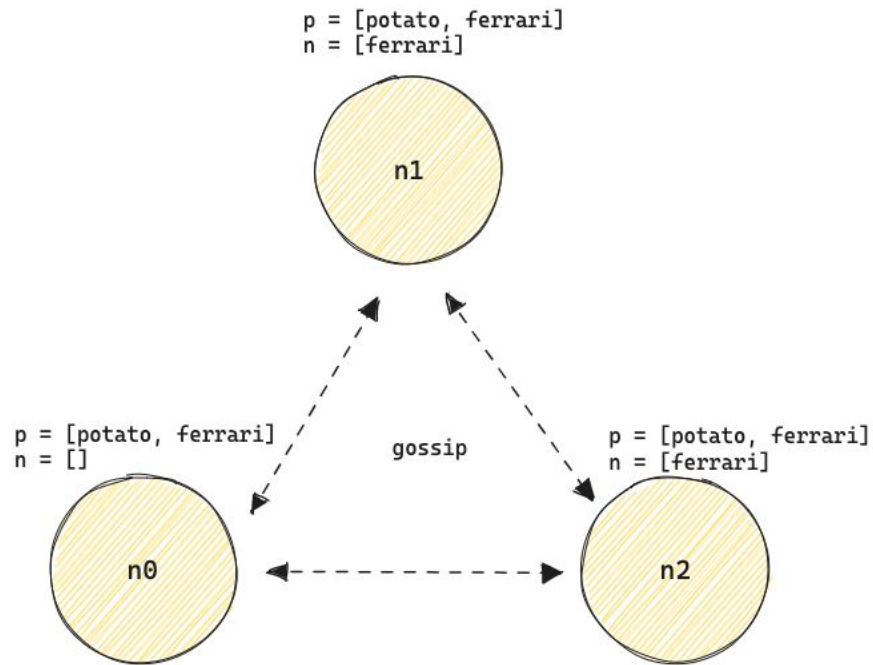
```
del(item T) { dels.insert(item) }
```

query:

```
read() []T {  
    return adds.difference(dels)  
}
```

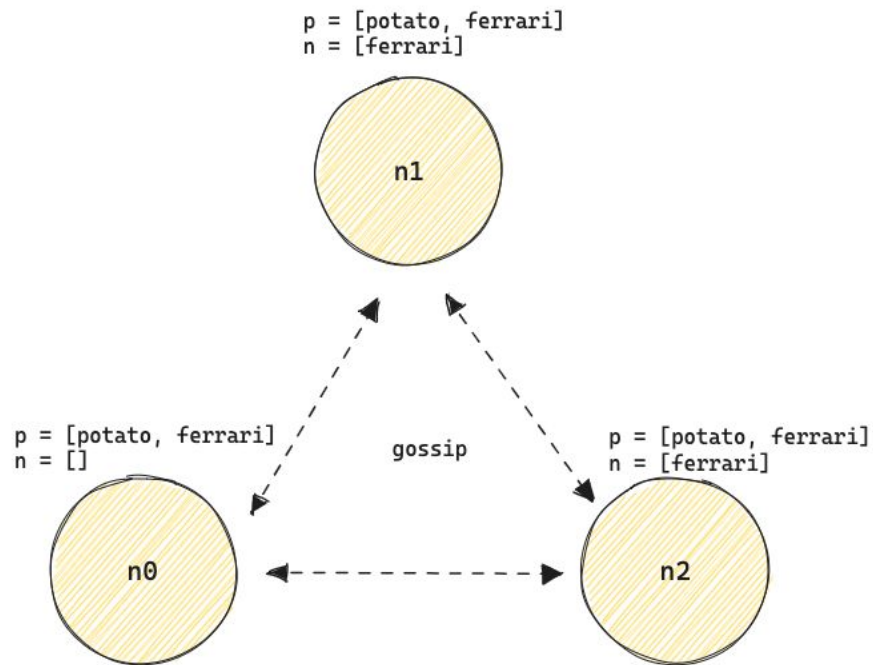
merge:

```
union(addItem, delItem) {  
    adds.union(addItem)  
    dels.union(delItem)  
}
```



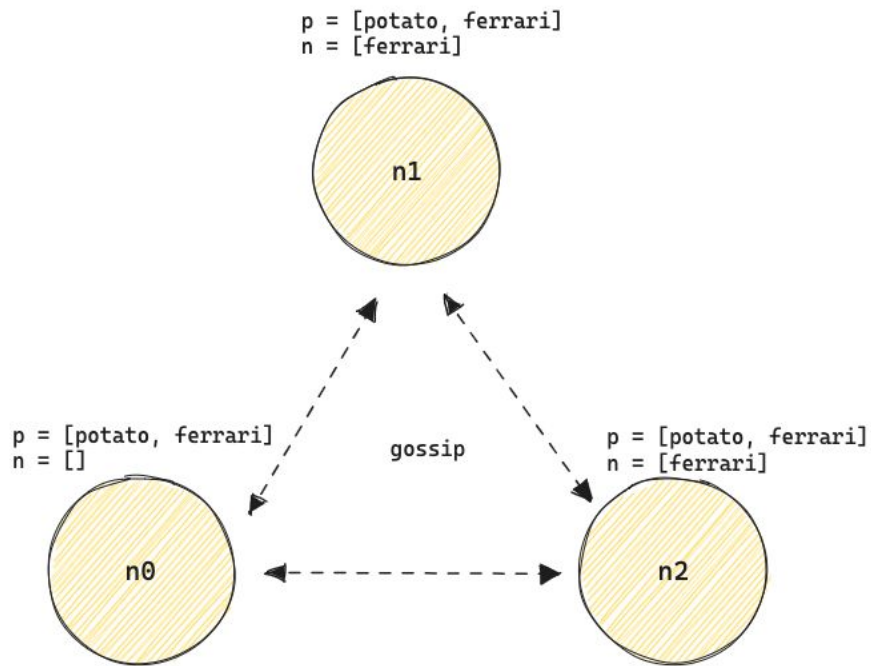
A Few Gotchas

- CRDTs provide mathematically sound guarantees for convergence.
- Or in other words, they provide guarantees for *liveness*.
- But this guarantee is only for updates. CRDTs provide no APIs or guarantees for *visibility* into the state (reads).
- No guarantees for safety when *reading* state!



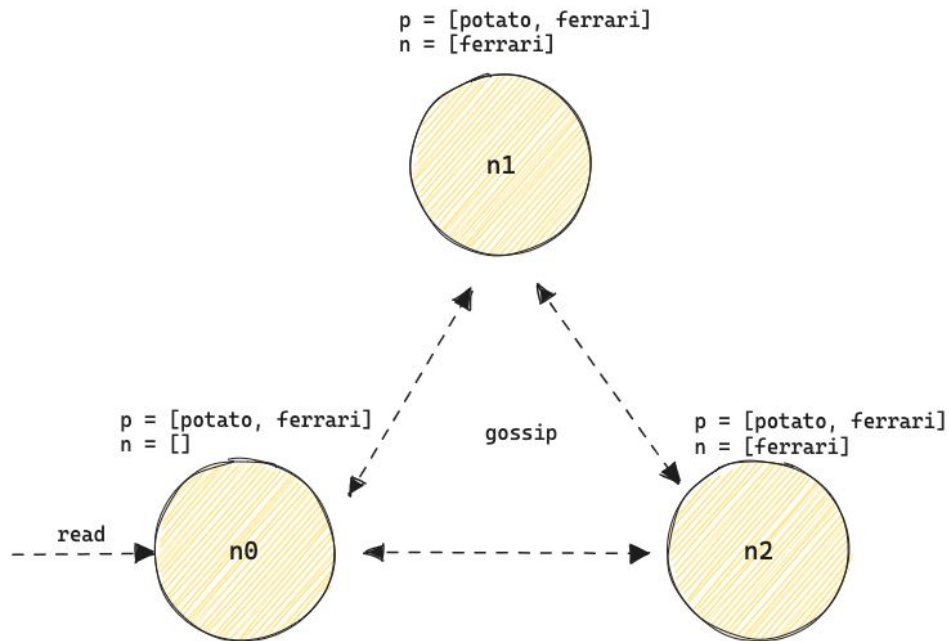
A Few Gotchas

- CRDTs provide mathematically sound guarantees for convergence.
- Or in other words, they provide guarantees for *liveness*.
- But this guarantee is only for updates. CRDTs provide no APIs or guarantees for *visibility* into the state (reads).
- No guarantees for safety when *reading* state!



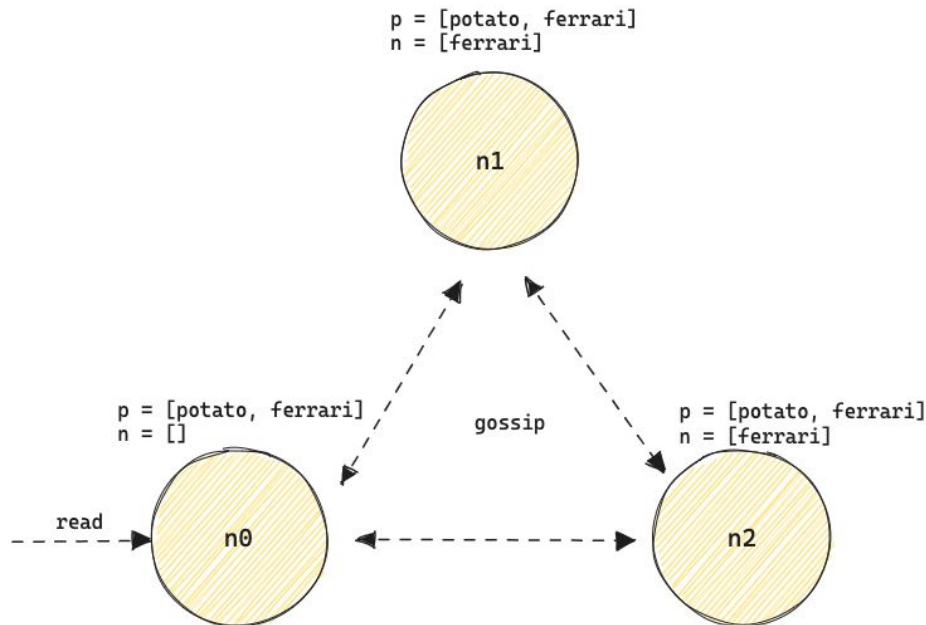
A Few Gotchas

- CRDTs provide mathematically sound guarantees for convergence.
- Or in other words, they provide guarantees for *liveness*.
- But this guarantee is only for updates. CRDTs provide no APIs or guarantees for *visibility* into the state (reads).
- No guarantees for safety when *reading* state!



A Few Gotchas

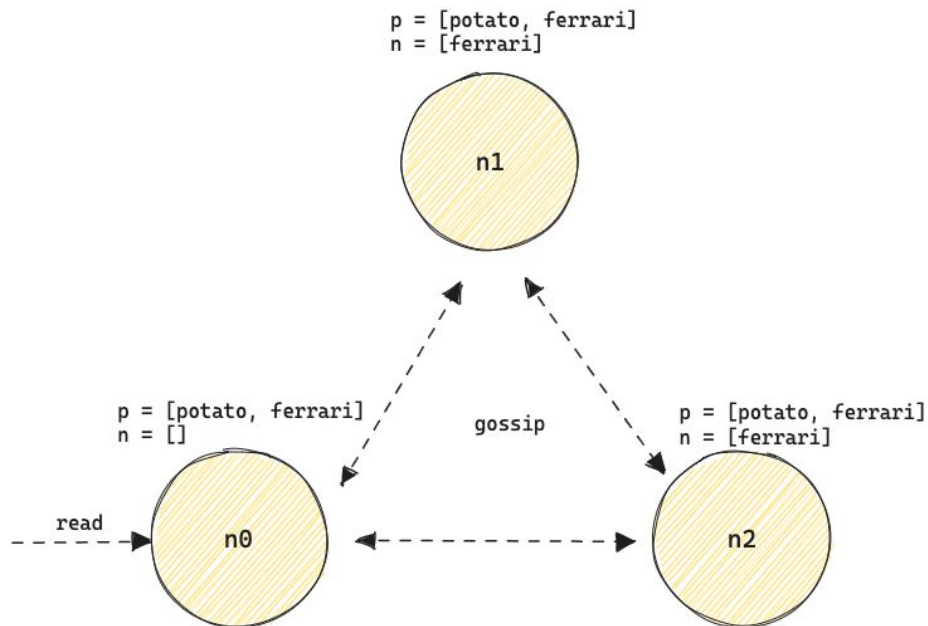
- CRDTs provide mathematically sound guarantees for convergence.
- Or in other words, they provide guarantees for *liveness*.
- But this guarantee is only for updates. CRDTs provide no APIs or guarantees for *visibility* into the state (reads).
- No guarantees for safety when *reading* state!



A Few Gotchas

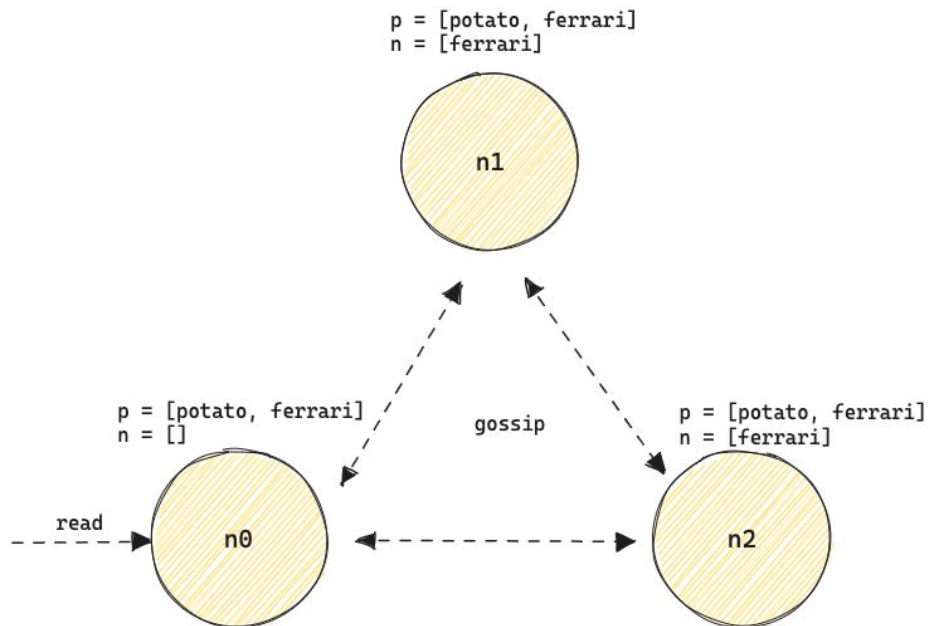
- How about we wait for all updates to arrive before processing

a checkout (read) request?



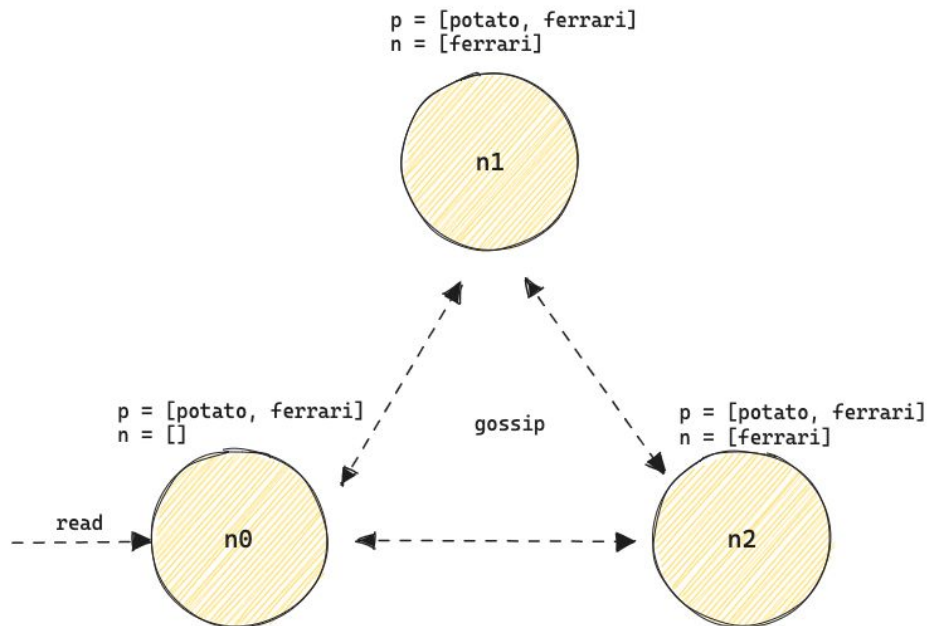
A Few Gotchas

- How about we wait for all updates to arrive before processing a checkout (`read`) request?
- Need to know what updates are present on other nodes.



A Few Gotchas

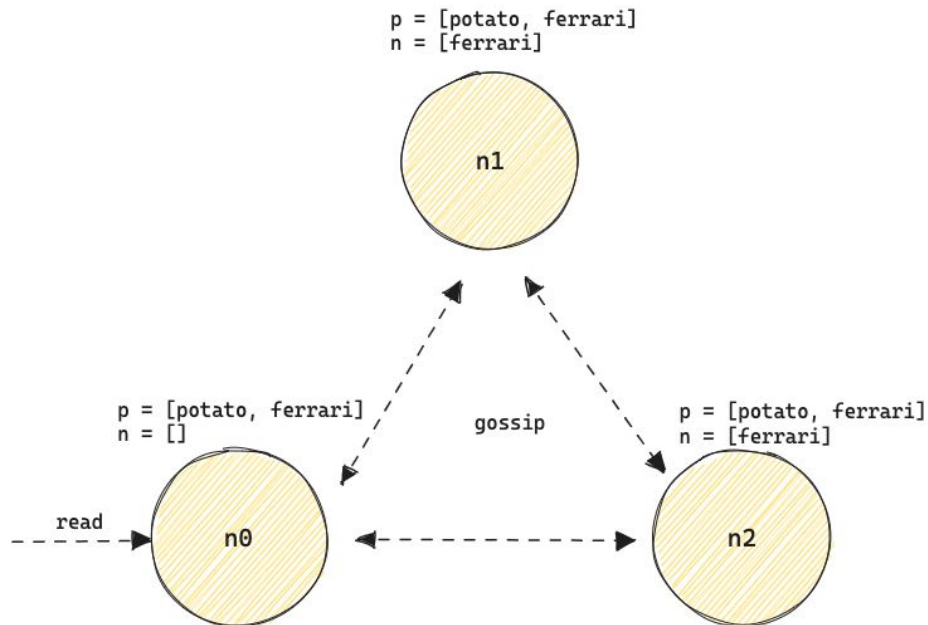
- How about we wait for all updates to arrive before processing a checkout (`read`) request?
- Need to know what updates are present on other nodes.
- Maybe we can ask other nodes?



A Few Gotchas

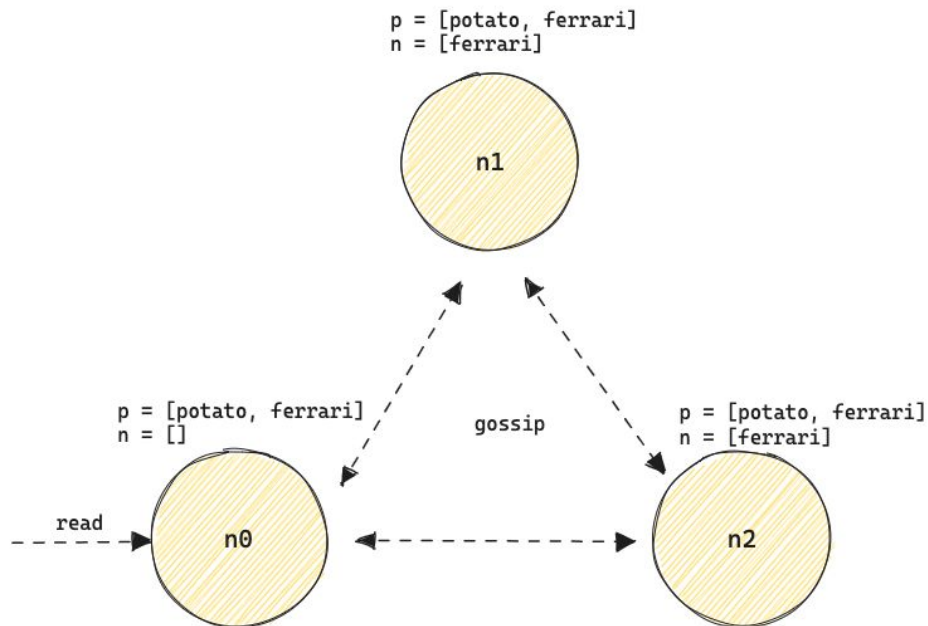
- How about we wait for all updates to arrive before processing a checkout (`read`) request?
- Need to know what updates are present on other nodes.
- Maybe we can ask other nodes?
- Hold on...

We're back in coordination land!



A Few Gotchas

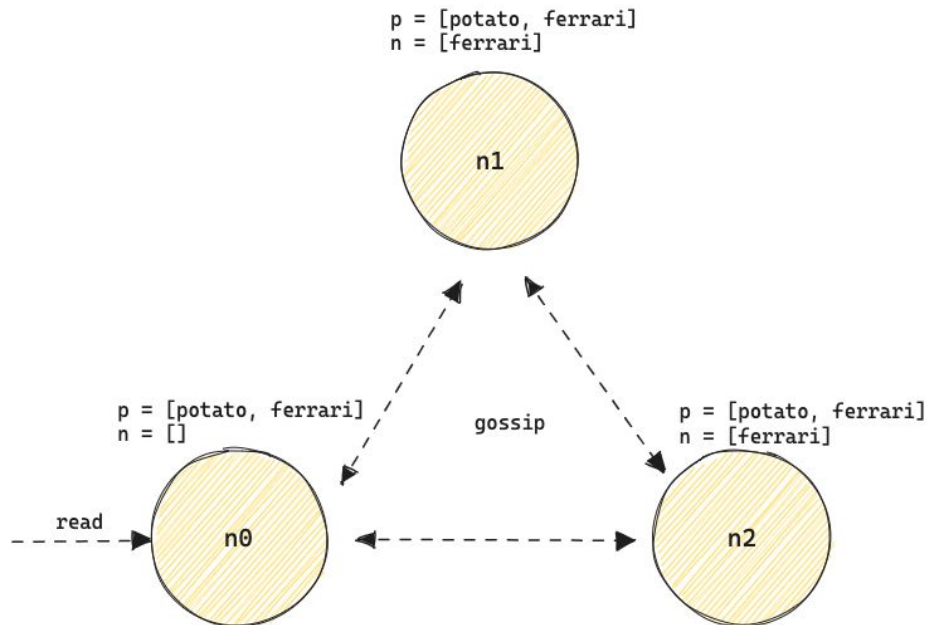
- CRDTs are guaranteed to be consistent as long as they are not observed.



A Few Gotchas

- CRDTs are guaranteed to be consistent as long as they are not observed.

CRDTs provide *Schrödinger Consistency Guarantees*



A Few Gotchas

- Why is that the case? Why did we end up back in coordination

land?

- Reads are ANNOYING!

- Reads don't *usually* commute with other operations.

```
del(ferrari) -> {potato} - {ferrari} !=
```

```
{potato, ferrari} - {} -> del(ferrari)
```

Building on Quicksand

Pat Helland
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
PHelland@Microsoft.com

Dave Campbell
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
DavidC@Microsoft.com

ABSTRACT

Reliable systems have always been built out of unreliable components [1]. Early on, the reliable components were small such as mirrored disks or ECC (Error Correcting Codes) in core memory. These systems were designed such that failures of these small components were transparent to the application. Later, the size of the unreliable components grew larger and semantic challenges crept into the application when failures occurred.

Fault tolerant algorithms comprise a set of idempotent sub-algorithms. Between these idempotent sub-algorithms, state is sent across the failure boundaries of the unreliable components. The failure of an unreliable component can then be tolerated as a takeover by a backup, which uses the last known state and drives forward with a retry of the idempotent sub-algorithm. Classically, this has been done in a linear fashion (i.e. one step at a time).

As the granularity of the unreliable component grows (from a mirrored disk to a system to a data center), the latency to communicate with a backup becomes unpalatable. This leads to a more relaxed model for fault tolerance. The primary system will acknowledge the work request and its actions *without* waiting to ensure that the backup is notified of the work. This improves the responsiveness of the system because the user is not delayed behind a slow interaction with the backup.

There are two implications of asynchronous state capture:

- 1) Everything promised by the primary is probabilistic. There is always a chance that an untimely failure shortly after the promise results in a backup proceeding without knowledge of the commitment. Hence, nothing is guaranteed!

2) Applications must assume eventual consistency [2]. Since

Keywords

Fault Tolerance, Eventual Consistency, Reconciliation, Loose Coupling, Transactions

1. Introduction

There is an interesting connection between fault tolerance, offlineable systems, and the need for application-based eventual consistency. As we attempt to run our large scale applications spread across many systems, we cannot afford the latency to wait for a backup system to remain in synch with the system actually performing the work. This causes the server systems to look increasingly like offlineable client applications in that they do not know the authoritative truth. In turn, these server-based applications are designed to record their intentions and allow the work to interleave and flow across the replicas. In a properly designed application, this results in system behavior that is acceptable to the business while being resilient to an increasing number of system failures.

This paper starts by examining the concepts of fault tolerance and posits an abstraction for thinking about fault tolerant systems. Next, section 3 examines how fault tolerant systems have historically provided the ability to transparently survive failures without special application consideration by using synchronous checkpointing to send the application state to a backup. In section 4, we begin to examine what happens when we cannot afford the latency associated with the synchronous checkpointing of state to the backup and, instead, allow the checkpointing of state to be asynchronous. Section 5 examines in much more depth the ways in which an application must be modified to be true to its

A Few Gotchas

- Why is that the case? Why did we end up back in coordination

land?

- Reads are ANNOYING!

- Reads don't *usually* commute with other operations.

```
del(ferrari) -> {potato, ferrari} - {ferrari} !=
```

```
{potato, ferrari} - {} -> del(ferrari)
```

We cannot reorder set difference (`read()`)!

Meaning we need to synchronize access.

Leading to need for coordination.

Building on Quicksand

Pat Helland
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
PHelland@Microsoft.com

Dave Campbell
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
DavidC@Microsoft.com

ABSTRACT

Reliable systems have always been built out of unreliable components [1]. Early on, the reliable components were small such as mirrored disks or ECC (Error Correcting Codes) in core memory. These systems were designed such that failures of these small components were transparent to the application. Later, the size of the unreliable components grew larger and semantic challenges crept into the application when failures occurred.

Fault tolerant algorithms comprise a set of idempotent sub-algorithms. Between these idempotent sub-algorithms, state is sent across the failure boundaries of the unreliable components. The failure of an unreliable component can then be tolerated as a takeover by a backup, which uses the last known state and drives forward with a retry of the idempotent sub-algorithm. Classically, this has been done in a linear fashion (i.e. one step at a time).

As the granularity of the unreliable component grows (from a mirrored disk to a system to a data center), the latency to communicate with a backup becomes unpalatable. This leads to a more relaxed model for fault tolerance. The primary system will acknowledge the work request and its actions *without* waiting to ensure that the backup is notified of the work. This improves the responsiveness of the system because the user is not delayed behind a slow interaction with the backup.

There are two implications of asynchronous state capture:

- 1) Everything promised by the primary is probabilistic. There is always a chance that an untimely failure shortly after the promise results in a backup proceeding without knowledge of the commitment. Hence, nothing is guaranteed!

2) Applications must assume eventual consistency [20]. Since

Keywords

Fault Tolerance, Eventual Consistency, Reconciliation, Loose Coupling, Transactions

1. Introduction

There is an interesting connection between fault tolerance, offlineable systems, and the need for application-based eventual consistency. As we attempt to run our large scale applications spread across many systems, we cannot afford the latency to wait for a backup system to remain in synch with the system actually performing the work. This causes the server systems to look increasingly like offlineable client applications in that they do not know the authoritative truth. In turn, these server-based applications are designed to record their intentions and allow the work to interleave and flow across the replicas. In a properly designed application, this results in system behavior that is acceptable to the business while being resilient to an increasing number of system failures.

This paper starts by examining the concepts of fault tolerance and posits an abstraction for thinking about fault tolerant systems. Next, section 3 examines how fault tolerant systems have historically provided the ability to transparently survive failures without special application consideration by using synchronous checkpointing to send the application state to a backup. In section 4, we begin to examine what happens when we cannot afford the latency associated with the synchronous checkpointing of state to the backup and, instead, allow the checkpointing of state to be asynchronous. Section 5 examines in much more depth the ways in which an application must be modified to be true to its

A Few Gotchas

- If we can somehow get that read to commute, we won't have to coordinate.

Pat Helland
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
PHelland@Microsoft.com

Building on Quicksand

Dave Campbell
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
DavidC@Microsoft.com

ABSTRACT

Reliable systems have always been built out of unreliable components [1]. Early on, the reliable components were small such as mirrored disks or ECC (Error Correcting Codes) in core memory. These systems were designed such that failures of these small components were transparent to the application. Later, the size of the unreliable components grew larger and semantic challenges crept into the application when failures occurred.

Fault tolerant algorithms comprise a set of idempotent sub-algorithms. Between these idempotent sub-algorithms, state is sent across the failure boundaries of the unreliable components. The failure of an unreliable component can then be tolerated as a takeover by a backup, which uses the last known state and drives forward with a retry of the idempotent sub-algorithm. Classically, this has been done in a linear fashion (i.e. one step at a time).

As the granularity of the unreliable component grows (from a mirrored disk to a system to a data center), the latency to communicate with a backup becomes unpalatable. This leads to a more relaxed model for fault tolerance. The primary system will acknowledge the work request and its actions *without* waiting to ensure that the backup is notified of the work. This improves the responsiveness of the system because the user is not delayed behind a slow interaction with the backup.

There are two implications of asynchronous state capture:

- 1) Everything promised by the primary is probabilistic. There is always a chance that an untimely failure shortly after the promise results in a backup proceeding without knowledge of the commitment. Hence, nothing is guaranteed!

2) Applications must never expect consistency [2]. Since

Keywords

Fault Tolerance, Eventual Consistency, Reconciliation, Loose Coupling, Transactions

1. Introduction

There is an interesting connection between fault tolerance, offlineable systems, and the need for application-based eventual consistency. As we attempt to run our large scale applications spread across many systems, we cannot afford the latency to wait for a backup system to remain in synch with the system actually performing the work. This causes the server systems to look increasingly like offlineable client applications in that they do not know the authoritative truth. In turn, these server-based applications are designed to record their intentions and allow the work to interleave and flow across the replicas. In a properly designed application, this results in system behavior that is acceptable to the business while being resilient to an increasing number of system failures.

This paper starts by examining the concepts of fault tolerance and posits an abstraction for thinking about fault tolerant systems. Next, section 3 examines how fault tolerant systems have historically provided the ability to transparently survive failures without special application consideration by using synchronous checkpointing to send the application state to a backup. In section 4, we begin to examine what happens when we cannot afford the latency associated with the synchronous checkpointing of state to the backup and, instead, allow the checkpointing of state to be asynchronous. Section 5 examines in much more depth the ways in which an application must be modified to be true to its

A Few Gotchas

- If we can somehow get that read to commute, we won't have to coordinate.
- Here's the thing... the reason it does not commute is because the output of our query (`read()`) is not stable.

Building on Quicksand

Pat Helland
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
PHelland@Microsoft.com

Dave Campbell
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
DavidC@Microsoft.com

ABSTRACT

Reliable systems have always been built out of unreliable components [1]. Early on, the reliable components were small such as mirrored disks or ECC (Error Correcting Codes) in core memory. These systems were designed such that failures of these small components were transparent to the application. Later, the size of the unreliable components grew larger and semantic challenges crept into the application when failures occurred.

Fault tolerant algorithms comprise a set of idempotent sub-algorithms. Between these idempotent sub-algorithms, state is sent across the failure boundaries of the unreliable components. The failure of an unreliable component can then be tolerated as a takeover by a backup, which uses the last known state and drives forward with a retry of the idempotent sub-algorithm. Classically, this has been done in a linear fashion (i.e. one step at a time).

As the granularity of the unreliable component grows (from a mirrored disk to a system to a data center), the latency to communicate with a backup becomes unpalatable. This leads to a more relaxed model for fault tolerance. The primary system will acknowledge the work request and its actions *without* waiting to ensure that the backup is notified of the work. This improves the responsiveness of the system because the user is not delayed behind a slow interaction with the backup.

There are two implications of asynchronous state capture:

- 1) Everything promised by the primary is probabilistic. There is always a chance that an untimely failure shortly after the promise results in a backup proceeding without knowledge of the commitment. Hence, nothing is guaranteed!

2) Applications must never expect consistency [20]. Since

Keywords

Fault Tolerance, Eventual Consistency, Reconciliation, Loose Coupling, Transactions

1. Introduction

There is an interesting connection between fault tolerance, offlineable systems, and the need for application-based eventual consistency. As we attempt to run our large scale applications spread across many systems, we cannot afford the latency to wait for a backup system to remain in synch with the system actually performing the work. This causes the server systems to look increasingly like offlineable client applications in that they do not know the authoritative truth. In turn, these server-based applications are designed to record their intentions and allow the work to interleave and flow across the replicas. In a properly designed application, this results in system behavior that is acceptable to the business while being resilient to an increasing number of system failures.

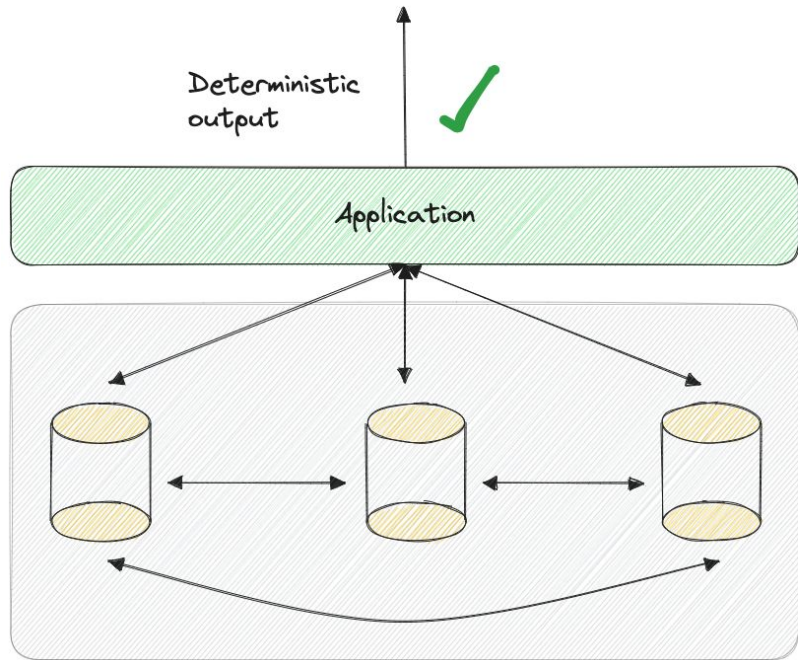
This paper starts by examining the concepts of fault tolerance and posits an abstraction for thinking about fault tolerant systems. Next, section 3 examines how fault tolerant systems have historically provided the ability to transparently survive failures without special application consideration by using synchronous checkpointing to send the application state to a backup. In section 4, we begin to examine what happens when we cannot afford the latency associated with the synchronous checkpointing of state to the backup and, instead, allow the checkpointing of state to be asynchronous. Section 5 examines in much more depth the ways in which an application must be modified to be true to its

A Few Gotchas

- If we can somehow get that read to commute, we won't have to coordinate.
- Here's the thing... the reason it does not commute is because the output of our query (`read()`) is not stable.
- Query can go back on what it outputted to be true at some point based on the updates it receives.

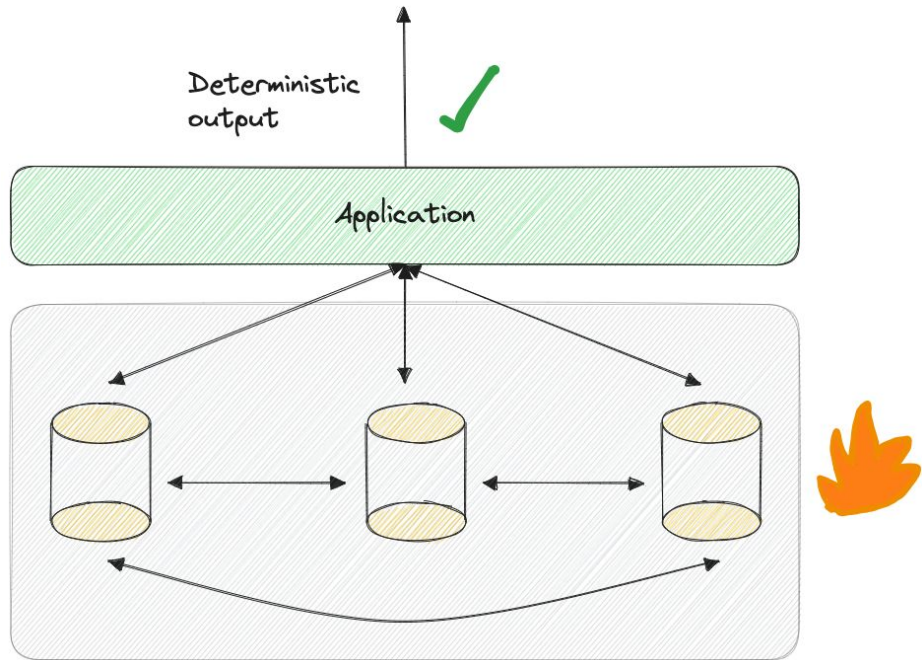
A Few Gotchas

- If we can somehow get that read to commute, we won't have to coordinate.
- Here's the thing... the reason it does not commute is because the output of our query (`read()`) is not stable.
- Query can go back on what it outputted to be true at some point based on the updates it receives.
- In other words, without coordination, we output not just stale but *false* information.



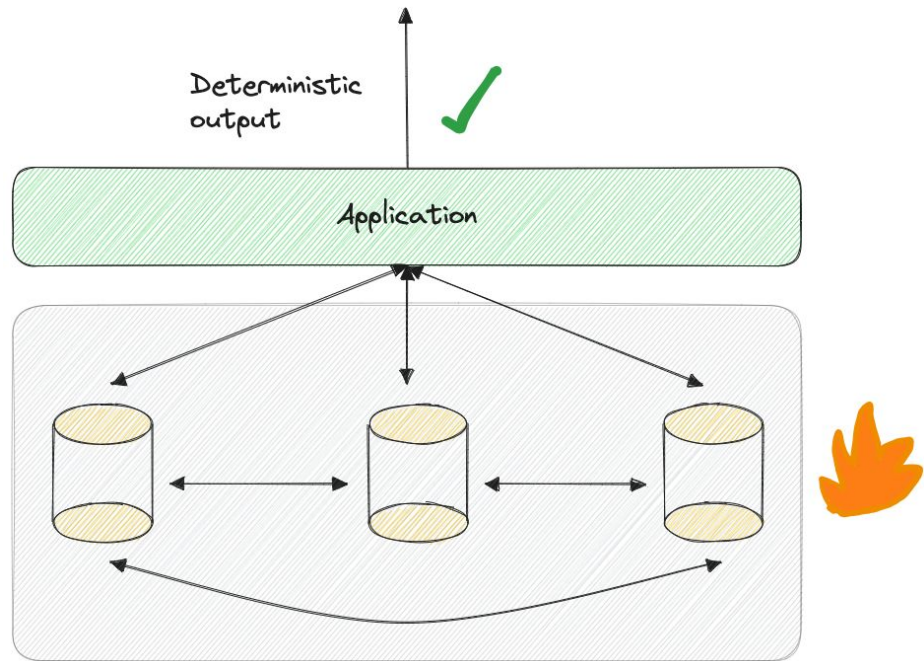
A Few Gotchas

- If outputs of a read query never retract what they have previously outputted, the query is stable.
- The worst case is you output (arbitrarily) stale information with new updates, but you will never output *false* information.



A Few Gotchas

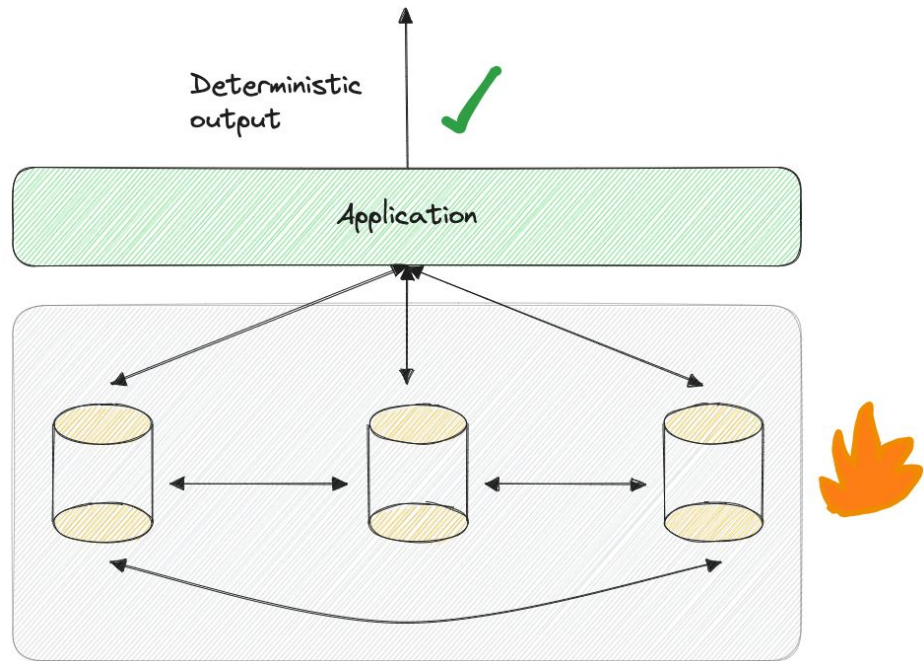
- This is starting to sound familiar...



A Few Gotchas

- This is starting to sound familiar...

MONOTONICITY TO THE RESCUE!



Keep CALM And CRDT On

- Here's the big idea...
- Along with a monotonic **op**, if a **query** is also monotonic, we can provide *liveness* AND *safety* guarantees for distributed execution over CRDTs.

Monotonic queries are queries whose output only ever "grows" with additional updates.

Keep CALM and CRDT On

“[...] can we develop a query model that makes it possible to precisely

define when execution on a single replica yields consistent results?”

Keep CALM and CRDT On

Shadaj Laddad*

University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*

University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano

University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung

University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks

University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein

University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

PVLDB Reference Format:

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks,

practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user’s perspective, the CRDT’s object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by

Keep CALM and CRDT On

“[...] can we develop a query model that makes it possible to precisely

define when execution on a single replica yields consistent results?”

- Helps identify what developers must reason about when using CRDTs.

Keep CALM and CRDT On

Shadaj Laddad*
University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*
University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano
University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks
University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein
University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

PVLDB Reference Format:

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks,

practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user’s perspective, the CRDT’s object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by

Keep CALM and CRDT On

“[...] can we develop a query model that makes it possible to precisely define when execution on a single replica yields consistent results?”

- Helps identify what developers must reason about when using CRDTs.
- Enables building data systems that *manage* CRDT replication and query execution, leading to stronger consistency guarantees.

Keep CALM and CRDT On

Shadaj Laddad*
University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*
University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano
University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks
University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein
University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

PVLDB Reference Format:

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks,

practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user’s perspective, the CRDT’s object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by

Towards A Query Model For CRDTs

Proposed query model for CRDTs:

Keep CALM and CRDT On

Shadaj Laddad*

University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*

University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano

University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung

University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks

University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein

University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

PVLDB Reference Format:

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks,

practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user's perspective, the CRDT's object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by

Towards A Query Model For CRDTs

Proposed query model for CRDTs:

- **Safety:** Queries should be sequentially consistent,

regardless of the replica at which they are evaluated.

Keep CALM and CRDT On

Shadaj Laddad*
University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*
University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano
University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks
University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein
University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

PVLDB Reference Format:

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks,

practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user's perspective, the CRDT's object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by

Towards A Query Model For CRDTs

Proposed query model for CRDTs:

- **Safety:** Queries should be sequentially consistent, regardless of the replica at which they are evaluated.
- **Efficiency:** Queries should be evaluated locally without coordination whenever possible.

Keep CALM and CRDT On

Shadaj Laddad*
University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*
University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano
University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks
University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein
University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

PVLDB Reference Format:

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks,

practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user's perspective, the CRDT's object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by

Towards A Query Model For CRDTs

Proposed query model for CRDTs:

- **Safety:** Queries should be sequentially consistent, regardless of the replica at which they are evaluated.
- **Efficiency:** Queries should be evaluated locally without coordination whenever possible.
- **Simplicity:** The query model should be easy for developers to reason about.

Keep CALM and CRDT On

Shadaj Laddad*
University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*
University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano
University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks
University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein
University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

PVLDB Reference Format:

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks,

practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user's perspective, the CRDT's object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by

Towards A Query Model For CRDTs

Example queries

EXAMPLE 2 (A BOOLEAN THRESHOLD QUERY OVER A GROW-ONLY SET). *One of the simplest possible CRDTs is the Grow-Only Set (G-Set) [51]. It is a set S with an operation that can add elements to S and a merge function that is the set union, $S_1 \cup S_2$.*

Say we want to determine whether the number of gift card purchases that are over \$100 in a set of transactions has exceeded 50 items (similar to the threshold functions in LVars [29]):

```
query suspicious_activity() {  
  if (cardinality(  
    txn for txn in state  
    if txn.type == "GIFTCARD" and txn.amount > 100  
  ]) > 50):  
    return true else ABORT;  
}
```

Towards A Query Model For CRDTs

Example queries

- Executing query on local replica will always produce a sequentially consistent result, even without coordination.

EXAMPLE 2 (A BOOLEAN THRESHOLD QUERY OVER A GROW-ONLY SET). *One of the simplest possible CRDTs is the Grow-Only Set (G-Set) [51]. It is a set S with an operation that can add elements to S and a merge function that is the set union, $S_1 \cup S_2$.*

Say we want to determine whether the number of gift card purchases that are over \$100 in a set of transactions has exceeded 50 items (similar to the threshold functions in LVars [29]):

```
query suspicious_activity() {  
  if (cardinality(  
    txn for txn in state  
    if txn.type == "GIFTCARD" and txn.amount > 100  
  ]) > 50):  
    return true else ABORT;  
}
```

Towards A Query Model For CRDTs

Example queries

- Executing query on local replica will always produce a sequentially consistent result, even without coordination.
- The true value of the query can never change once observed, even with additional updates.

EXAMPLE 2 (A BOOLEAN THRESHOLD QUERY OVER A GROW-ONLY SET). *One of the simplest possible CRDTs is the Grow-Only Set (G-Set) [51]. It is a set S with an operation that can add elements to S and a merge function that is the set union, $S_1 \cup S_2$.*

Say we want to determine whether the number of gift card purchases that are over \$100 in a set of transactions has exceeded 50 items (similar to the threshold functions in LVars [29]):

```
query suspicious_activity() {  
  if (cardinality(  
    txn for txn in state  
    if txn.type == "GIFTCARD" and txn.amount > 100  
  ]) > 50):  
    return true else ABORT;  
}
```

Towards A Query Model For CRDTs

Example queries

- Executing query on local replica will always produce a sequentially consistent result, even without coordination.
- The true value of the query can never change once observed, even with additional updates.
- Local state + some updates = global state

EXAMPLE 2 (A BOOLEAN THRESHOLD QUERY OVER A GROW-ONLY SET). *One of the simplest possible CRDTs is the Grow-Only Set (G-Set) [51]. It is a set S with an operation that can add elements to S and a merge function that is the set union, $S_1 \cup S_2$.*

Say we want to determine whether the number of gift card purchases that are over \$100 in a set of transactions has exceeded 50 items (similar to the threshold functions in LVars [29]):

```
query suspicious_activity() {  
  if (cardinality(  
    txn for txn in state  
    if txn.type == "GIFTCARD" and txn.amount > 100  
  ]) > 50):  
    return true else ABORT;  
}
```

Towards A Query Model For CRDTs

Example queries

- Executing query on local replica will always produce a sequentially consistent result, even without coordination.
- The true value of the query can never change once observed, even with additional updates.
- Local state + some updates = global state
- Most importantly: you might read stale information, but you will *never* read incorrect information.

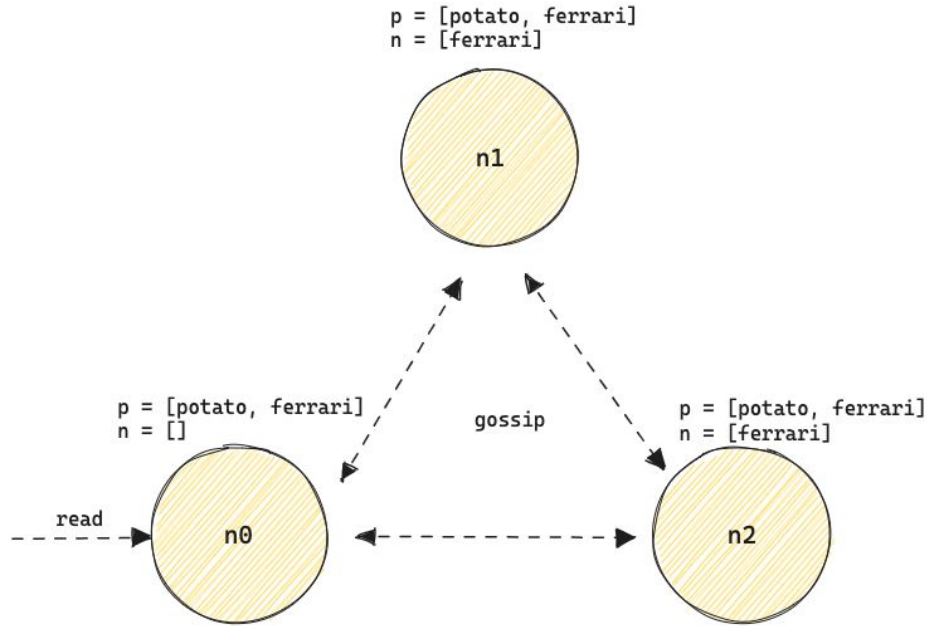
EXAMPLE 2 (A BOOLEAN THRESHOLD QUERY OVER A GROW-ONLY SET). *One of the simplest possible CRDTs is the Grow-Only Set (G-Set) [51]. It is a set S with an operation that can add elements to S and a merge function that is the set union, $S_1 \cup S_2$.*

Say we want to determine whether the number of gift card purchases that are over \$100 in a set of transactions has exceeded 50 items (similar to the threshold functions in LVars [29]):

```
query suspicious_activity() {  
  if (cardinality(  
    txn for txn in state  
    if txn.type == "GIFTCARD" and txn.amount > 100  
  ]) > 50):  
    return true else ABORT;  
}
```


Towards A Query Model For CR

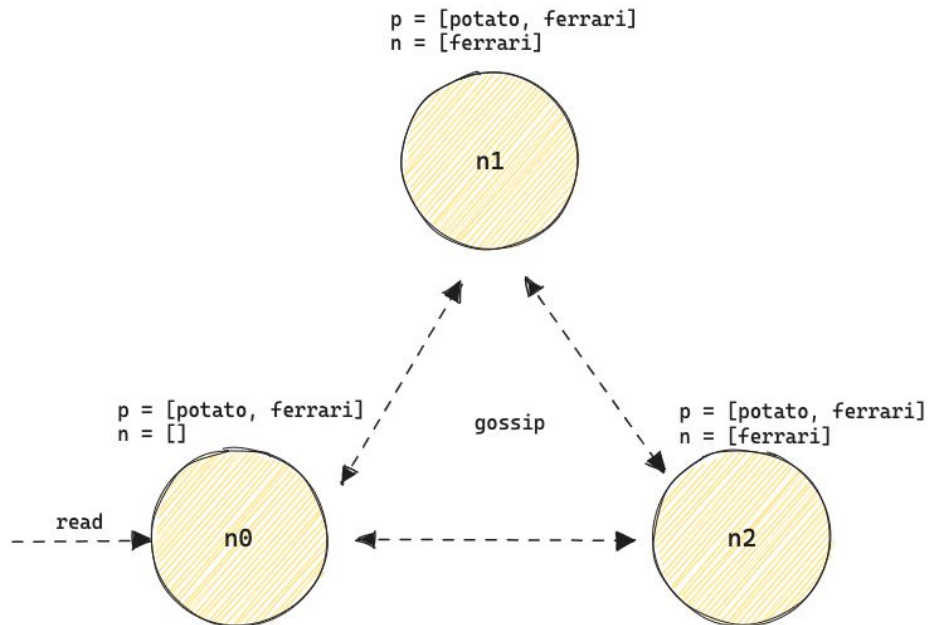
Example queries



Towards A Query Model For CR

Example queries

- As we saw, we cannot do away with coordination in this case.
- *Stale information is also incorrect information.*



Towards A Query Model For CRDTs

Remember how we said MONOTONICITY TO THE RESCUE?

Towards A Query Model For CRDTs

- The CALM theorem originally is framed for logic programs.
- It applies perfectly well to queries over CRDTs as well!
- We can define a monotone query as any whose output is monotone with respect to ordering of the CRDT.

Towards A Query Model For CRDTs

- The CALM theorem originally is framed for logic programs.
- It applies perfectly well to queries over CRDTs as well!
- We can define a monotone query as any whose output is monotone with respect to ordering of the CRDT.

“By the CALM Theorem, monotone queries over CRDTs are exactly the queries that only need a local view of the system to be correct!”

Towards A Query Model For CRDTs

“By the CALM Theorem, monotone queries over CRDTs are exactly the queries that only need a local view of the system to be correct!”

- Not just that, CALM tells that it is *only* monotone queries that can satisfy this criteria of coordination avoidance.
- Monotone queries meet all criteria of our good query model.

Towards A Query Model For CRDTs

“By the CALM Theorem, monotone queries over CRDTs are exactly the queries that only need a local view of the system to be correct!”

- Just as monotonic functions compose, monotonic queries compose too! Super powerful.
- Field of monotone queries is large - 4 of the 5 relational algebra operators are monotone.

Towards A Query Model For CRDTs

“By the CALM Theorem, monotone queries over CRDTs are exactly the queries that only need a local view of the system to be correct!”

- Very simple query model for developers to reason about.
- Understanding definition of CRDTs requires understanding monotonicity for state updates.
- Reasonable for developers to extend this reasoning to queries as well.
- If SQL is used, monotone queries can be syntactically identified – can leverage developer tooling here.

Towards A Query Model For CRDTs

But what about non-monotone queries? Not all business logic can be expressed monotonically.

Towards A Query Model For CRDTs

But what about non-monotone queries? Not all business logic can be expressed monotonically.

- Answer is simple – coordinate!
- However, coordination as well is improved upon here.
- All update operations commute, you need to order *sets* of updates, not *sequences* of them.
- We only care about which updates have arrived, and not their order.
- Contrast with Paxos or Raft, which enforces everyone, everywhere sees the same order no matter what.

Towards A Query Model For CRDTs

TL;DR – if the query you make against a CRDT is monotone, you can execute it safely locally without coordination.

If it is not monotone, you will need to coordinate.

Towards A Query Model For CRDTs

Next step: need to map query model to a practical language.

- Need a rich expressions that can manipulate CRDT stat (lattice structures).
- And syntax that is easily understood and can convey when a query is monotone or not (to both humans and computers).

Towards A Query Model For CRDTs

Next step: need to map query model to a practical language.

- Need a rich expressions that can manipulate CRDT stat (lattice structures).
- And syntax that is easily understood and can convey when a query is monotone or not (to both humans and computers).

A dialect of something already familiar to developers: SQL!

Towards A Query Model For CRDTs

Next step: need to map query model to a practical language.

You can make use of existing proofs in relational algebra:

- Need a rich expressions that can manipulate CRDT stat (lattice structures).
- And syntax that is easily understood and can convey when a query is monotone or not (to both humans and computers).

“If Q is a `SELECT-FROM-WHERE` query, Q is monotone.”

A dialect of something already familiar to developers: SQL!

Towards A Query Model For CRDTs

- This was our API previously.
- But with a query model and a query language...
- **op**: Clients use this to modify the state of the CRDT. Must be monotonic.
- **query**: Does not modify state, only returns some result that might depend on state.
- **merge**: Takes a value, merges it with existing state and produces new state. Must be Associative, Commutative and Idempotent (ACI).

Towards A Query Model For CRDTs

- This was our API previously.
- But with a query model and a query language, we no longer have a pre-defined set of queries.
- **op**: Clients use this to modify the state of the CRDT. Must be monotonic.
- **query**: Does not modify state, only returns some result that might depend on state.
- **merge**: Takes a value, merges it with existing state and produces new state. Must be Associative, Commutative and Idempotent (ACI).

Building Data Management Systems For CRDTs

- With a query model and language, queries are just
interfaces to the actual datastore.

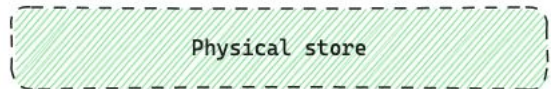
Building Data Management Systems For CRDTs

- With a query model and language, queries are just interfaces to the actual datastore.

“we propose a shift in perspective from an object-oriented view of CRDTs to a database view of them: breaking CRDTs up into a query model and a data store that separates their logical and physical representations.”

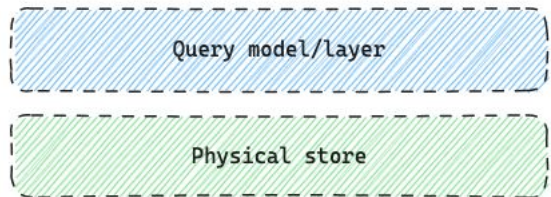
Building Data Management Systems For CRDTs

- Physical representation of data.



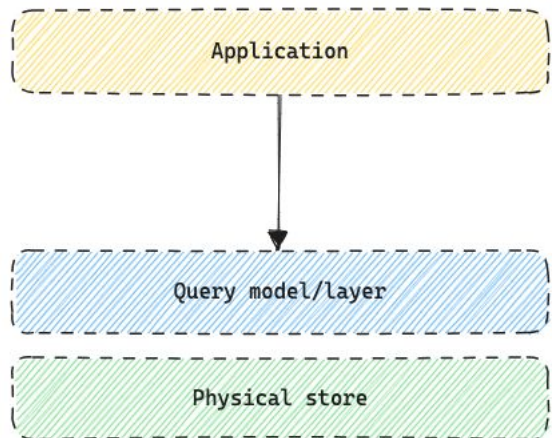
Building Data Management Systems For CRDTs

- Physical representation of data.
- Along with our query model.



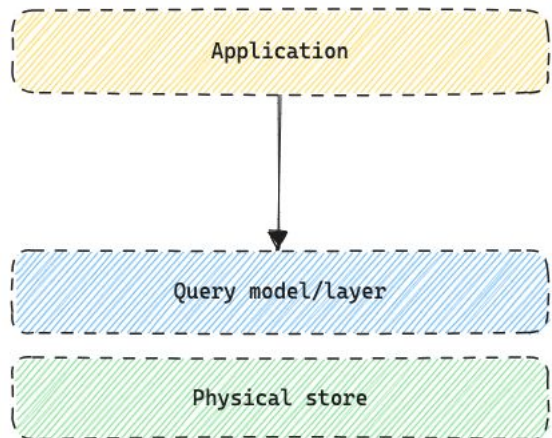
Building Data Management Systems For CRDTs

- Physical representation of data.
- Along with our query model.
- We have our application which can then communicate over the network – like any other data store deployed as a service.



Building Data Management Systems For CRDTs

- Physical representation of data.
- Along with our query model.
- We have our application which can then communicate over the network – like any other data store deployed as a service.



“We believe that this approach can both increase the ease of use of CRDTs, by shifting the responsibility of reasoning about consistency to the store, and improve the efficiency of applications built on CRDTs, since data stores can make optimization decisions based on the dynamic workload.”

“But I Reallllly Want A Non-monotone Query”

- From our query model – non-monotone queries need coordination in order to execute safely. Does that mean I accept my fate of high latencies and coordination bottlenecks?

“But I Reallllly Want A Non-monotone Query”

- From our query model – non-monotone queries need

coordination in order to execute safely. Does that mean I

accept my fate of high latencies and coordination

bottlenecks?

- Yes but also no. “Pre-fetch” and “pre-coordinate”.
- Sometimes you might just want weakly consistent systems, don't bother with coordination in any case here then.


“But I Reallllly Want A Non-monotone Query”

- Well... I'm not sure I want weak consistency, if only there

was a way for me to analyze just how eventual, this
eventual consistency is and understand how to better
program against it.

“But I Reallllly Want A Non-monotone Query”

- Well... I’m not sure I want weak consistency, if only there was a way for me to analyze just how eventual, this eventual consistency is and understand how to better program against it.



ACM Queue

DATABASES

Eventual Consistency Today: Limitations, Extensions, and Beyond

How can applications be built on eventually consistent infrastructure given no guarantee of safety?

Peter Bailis and Ali Ghodsi, UC Berkeley

In a July 2000 conference keynote, Eric Brewer, now VP of engineering at Google and a professor at the University of California, Berkeley, introduced the concept of eventual consistency. He argued that the partition tolerance of the first Internet services, such as Usenet, were architected for availability, not consistency. This is a particularly relevant point for today's Internet services, which are often built on weaker models—those that guarantee availability and consistency, but not partition tolerance. Eventual consistency updates are made to a single master, which then propagates the updates to the replicas. This is a particularly relevant point for today's Internet services, which are often built on weaker models—those that guarantee availability and consistency, but not partition tolerance. Yet, despite this approach, businesses are built on the assumption that the system will eventually return to a consistent state.

Probabilistically Bounded Staleness for Practical Partial Quorums

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, Ion Stoica
University of California, Berkeley
(pbailis, shivaram, franklin, hellerstein, istoica)@cs.berkeley.edu

All good ideas arrive by chance.—Max Ernst

ABSTRACT

Data store replication results in a fundamental trade-off between operation latency and data consistency. In this paper, we examine this trade-off in the context of quorum-replicated data stores. Under partial, or non-strict quorum replication, a data store waits for responses from a subset of replicas before answering a query, without guaranteeing that read and write replica sets intersect. As deployed in practice, these configurations provide only basic eventual consistency guarantees, with no limit to the recency of data returned. However, anecdotally, partial quorums are often “good enough” for practitioners given their latency benefits. In this work, we explain why partial quorums are regularly acceptable in practice, analyzing both the staleness of data they return and the latency benefits they offer. We introduce Probabilistically Bounded Staleness (PBS) consistency, which provides expected bounds on staleness with respect to both versions and wall clock time. We derive a closed-form solution for versioned staleness as well as model real-time staleness for representative Dynamo-style systems under internet-scale production workloads. Using PBS, we measure the latency-consistency trade-off for partial quorum systems. We quantitatively demonstrate how eventually consistent systems frequently return consistent data within tens of milliseconds while offering

39, 55]. However, eventually consistent systems make no guarantees on the staleness (recency in terms of versions written) of data items returned except that the system will “eventually” return the most recent version in the absence of new writes [61].

This latency-consistency trade-off inherent in distributed data stores has significant consequences for application design [6]. Low latency is critical for a large class of applications [56]. For example, at Amazon, 100 ms of additional latency resulted in a 1% drop in sales [44], while 500 ms of additional latency in Google's search resulted in a corresponding 20% decrease in traffic [45]. At scale, increased latencies correspond to large amounts of lost revenue, but lowering latency has a consistency cost: contacting fewer replicas for each request typically weakens the guarantees on returned data. Programs can often tolerate weak consistency by employing careful design patterns such as compensation (e.g., memories, guesses, and apologies) [33] and by using associative and commutative operations (e.g., timelines, logs, and notifications) [12]. However, potentially *unbounded* staleness (as in eventual consistency) poses significant challenges and is undesirable in practice.

1.1 Practical Partial Quorums

In this work, we examine the latency-consistency trade-off in the context of quorum-replicated data stores. Quorum systems ensure strong consistency across reads and writes to replicas by ensuring that read and write replica sets overlap. However, maintaining a

Resources

- [Main Paper 1] [Keep CALM And CRDT On](#)
- [Main Paper 2] [Keeping CALM: When Distributed Consistency Is Easy](#)
- [Paper] [Coordination Avoidance In Database Systems](#)
- [Paper] [Anna: A KVS For Any Scale](#)
- CRDTs
 - [Original Paper] [Conflict Free Replicated Data Types](#)
 - [Paper] [CRDTs: An Overview](#) (thanks to Lewis Campbell [[@LewisCTech](#)] for this resource!)
 - [Talk] [A CRDT Primer: Defanging Order Theory](#)
 - [Talk] [Strong Eventual Consistency and CRDTs](#)
 - [Talk] [Encapsulating replication, high concurrency and consistency with CRDTs](#)
 - [Code] [Implementations of a few CRDTs tested against Jepsen, written in Go](#)
- [Paper] [How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor](#)
- [Paper] [Building On Quicksand](#)
- [ACM Queue] [Eventual Consistency Today: Limitations, Extensions, and Beyond](#)
- [Talk, Paper] [PBS: Probabilistically Bounded Staleness](#)

Acknowledgements

Thank you [Conor Power](#) (an author of the Keep CALM And CRDT On paper) for helping answer some of my questions, and in great detail!

Thank you!

<https://nonmonotonic.dev/>