We hear you like PAPERS

INES SOMBRA

★ ★ ★

@Randommood

CAITIE MCCAFFREY

★ ★ ★

@Caitie

DISTRIBUTED Systems

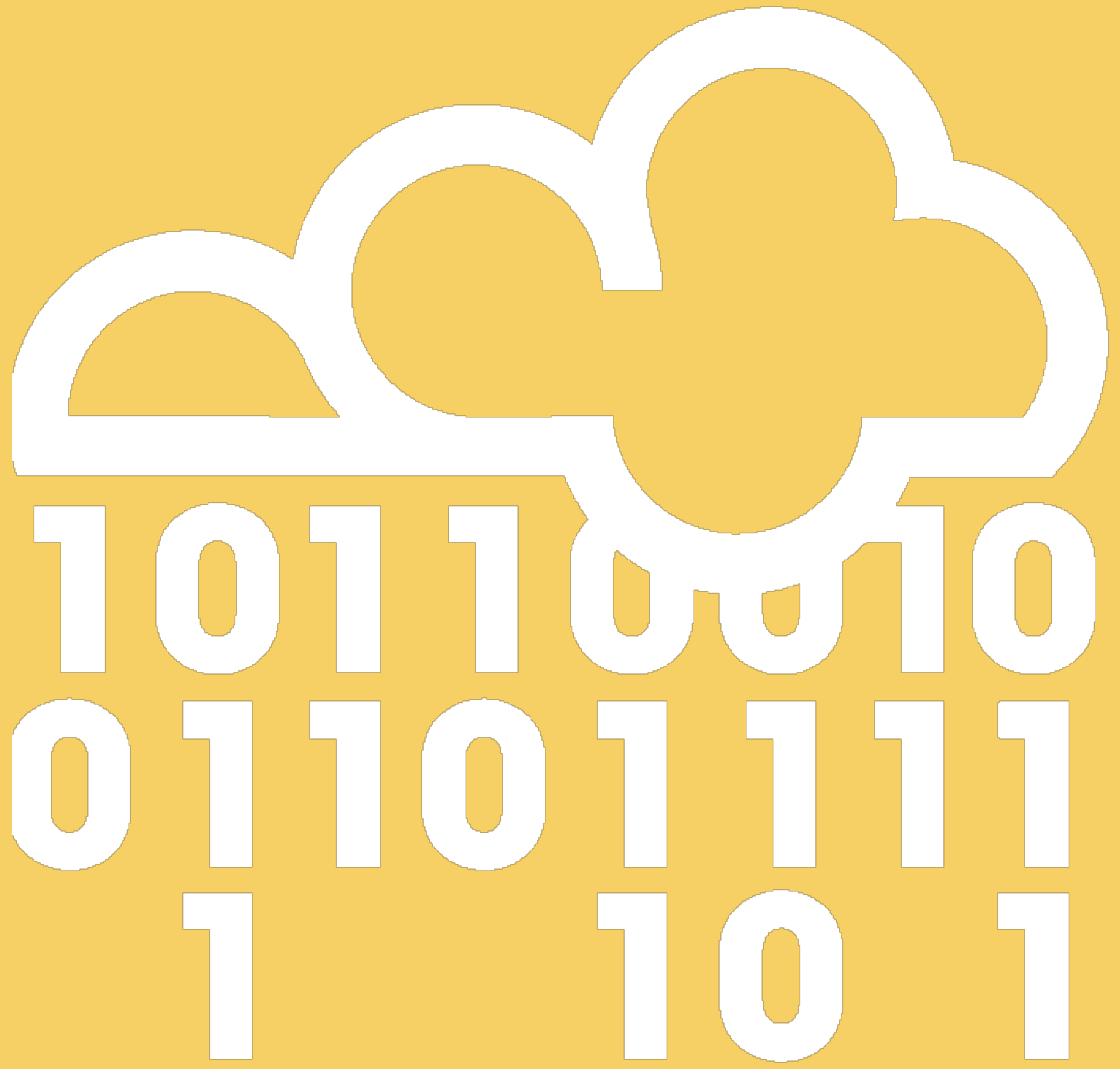
ACADEMIC Papers

OUR *Journey today*

EVENTUAL CONSISTENCY

SYSTEM VERIFICATION

EVENTUAL Consistency



THINKING *Consistency*

1983	1995	2002
★	★	★
Detection of Mutual Inconsistency in Distributed Systems	Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System	Brewer's conjecture & the feasibility of consistent, available, partition-tolerant web services

THINKING *Consistency*

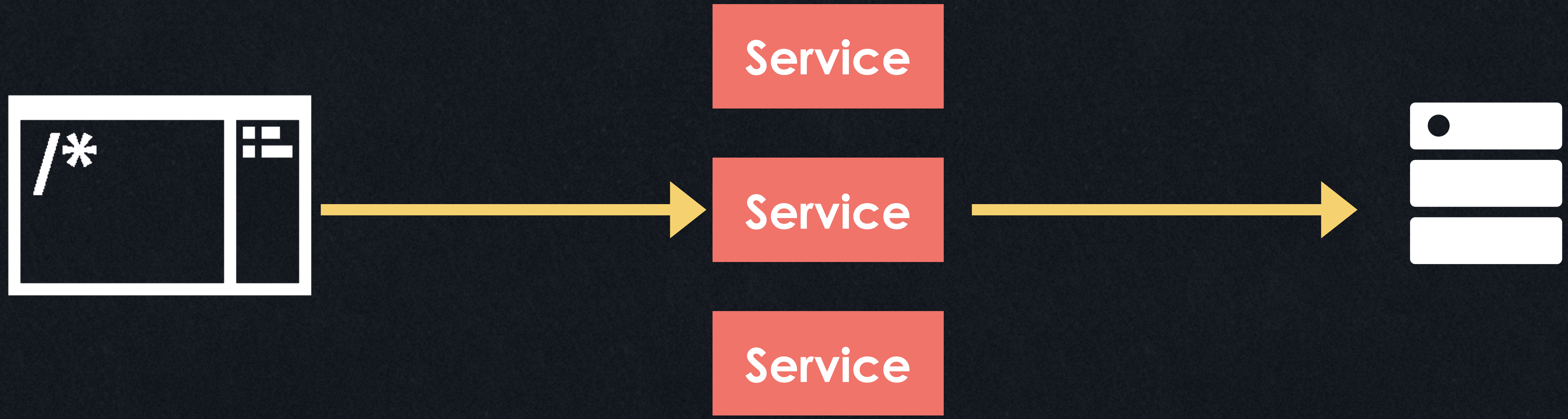
2011 ★

Conflict-free replicated Data Types

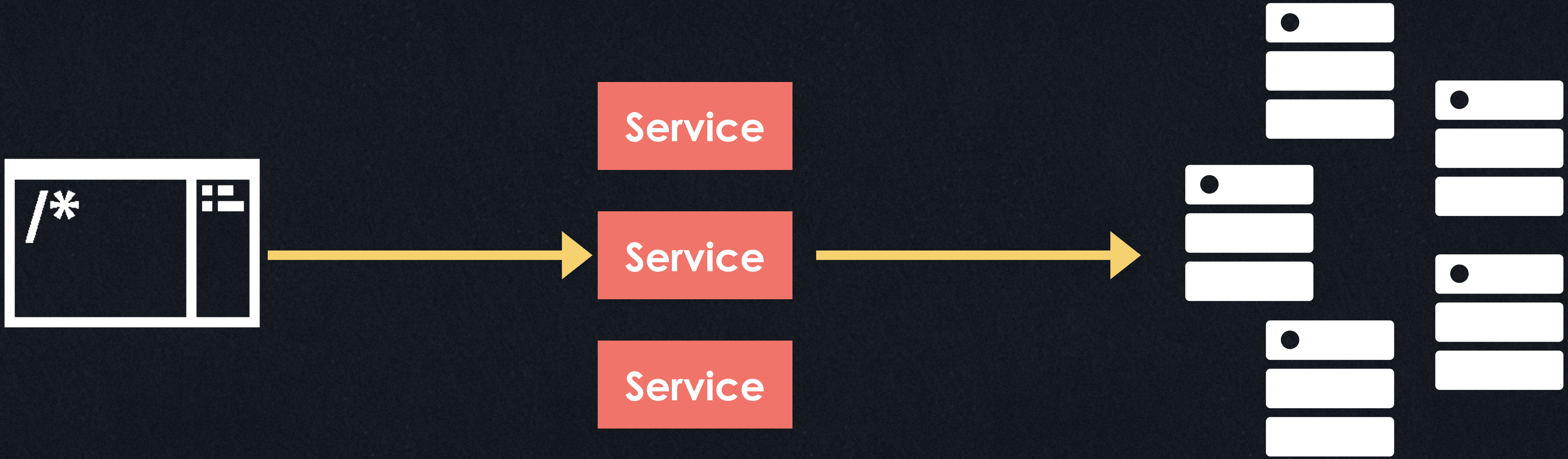
2015 ★

Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity

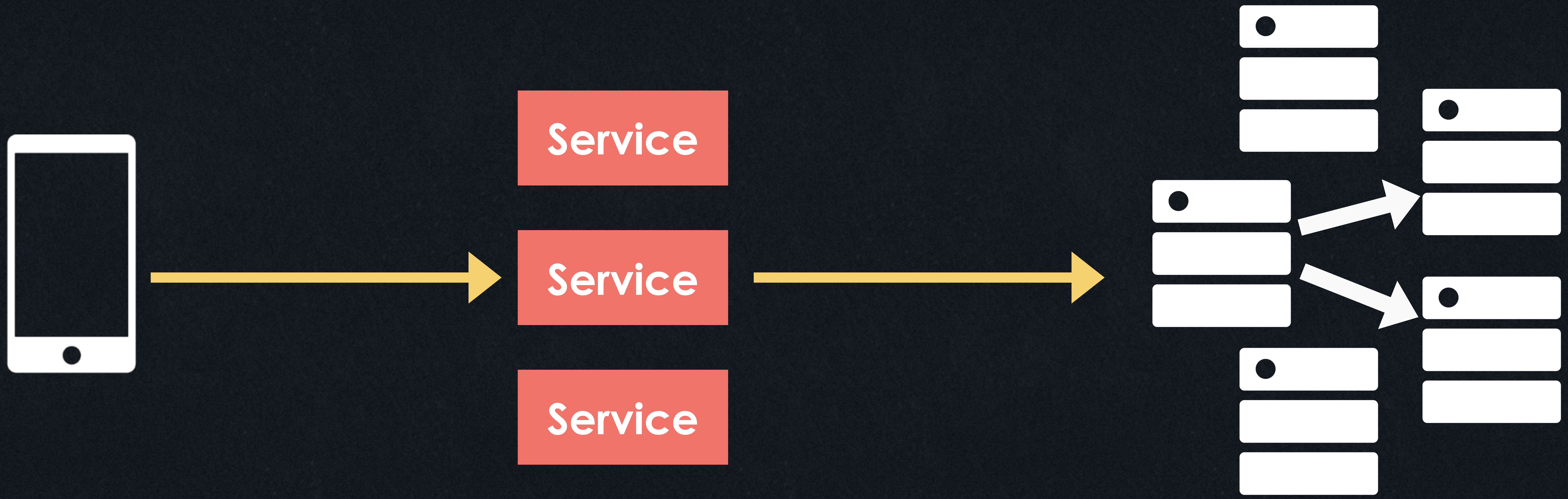
APPLICATIONS *Before*



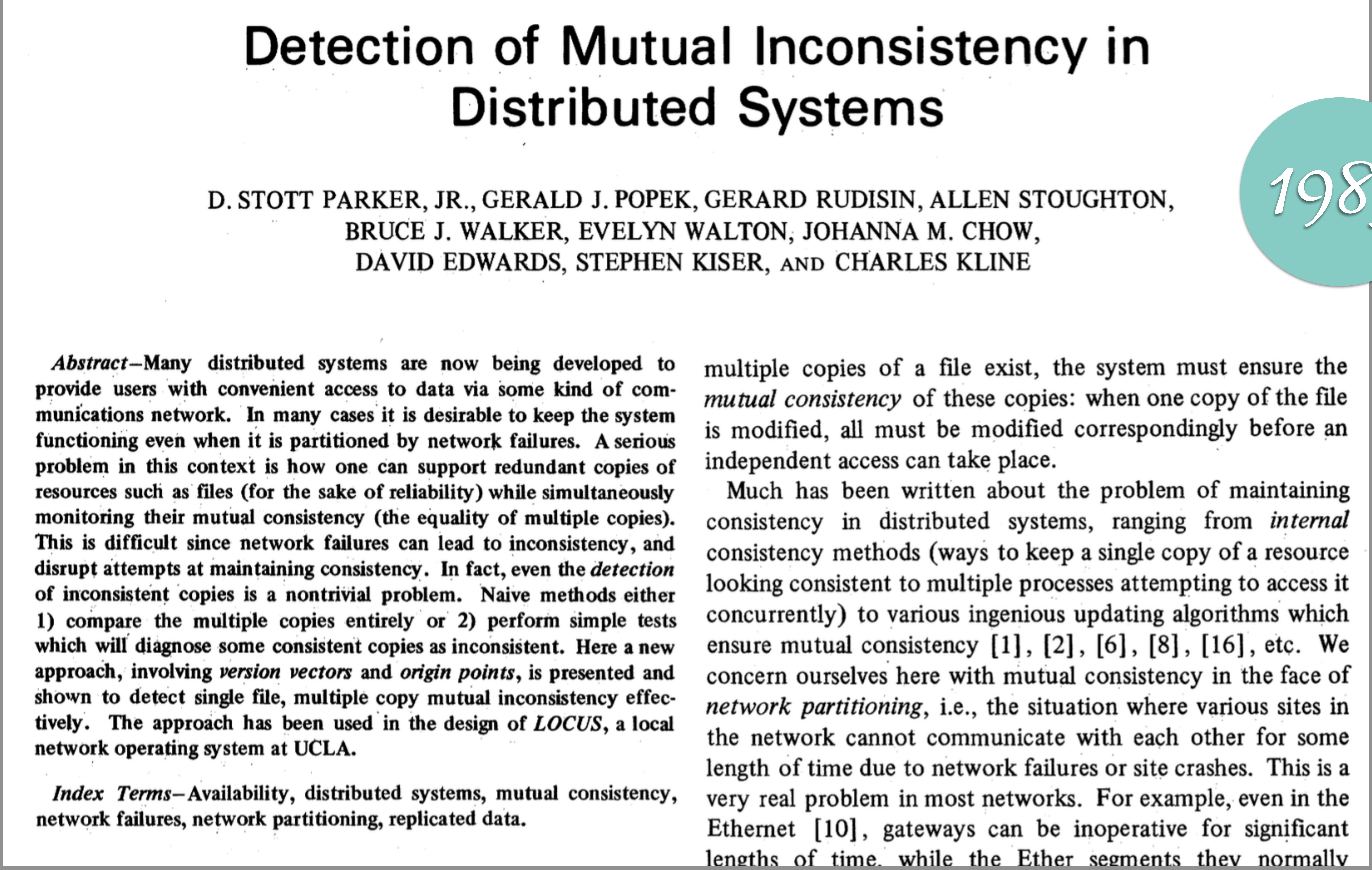
APPLICATIONS *Before*



APPLICATIONS *Now*



HIGH AVAILABILITY



Detection of Mutual Inconsistency in Distributed Systems

1983

D. STOTT PARKER, JR., GERALD J. POPEK, GERARD RUDISIN, ALLEN STOUGHTON, BRUCE J. WALKER, EVELYN WALTON, JOHANNA M. CHOW, DAVID EDWARDS, STEPHEN KISER, AND CHARLES KLINE

Abstract–Many distributed systems are now being developed to provide users with convenient access to data via some kind of communications network. In many cases it is desirable to keep the system functioning even when it is partitioned by network failures. A serious problem in this context is how one can support redundant copies of resources such as files (for the sake of reliability) while simultaneously monitoring their mutual consistency (the equality of multiple copies). This is difficult since network failures can lead to inconsistency, and disrupt attempts at maintaining consistency. In fact, even the *detection* of inconsistent copies is a nontrivial problem. Naive methods either 1) compare the multiple copies entirely or 2) perform simple tests which will diagnose some consistent copies as inconsistent. Here a new approach, involving *version vectors* and *origin points*, is presented and shown to detect single file, multiple copy mutual inconsistency effectively. The approach has been used in the design of *LOCUS*, a local network operating system at UCLA.

Index Terms–Availability, distributed systems, mutual consistency, network failures, network partitioning, replicated data.

multiple copies of a file exist, the system must ensure the *mutual consistency* of these copies: when one copy of the file is modified, all must be modified correspondingly before an independent access can take place.

Much has been written about the problem of maintaining consistency in distributed systems, ranging from *internal* consistency methods (ways to keep a single copy of a resource looking consistent to multiple processes attempting to access it concurrently) to various ingenious updating algorithms which ensure mutual consistency [1], [2], [6], [8], [16], etc. We concern ourselves here with mutual consistency in the face of *network partitioning*, i.e., the situation where various sites in the network cannot communicate with each other for some length of time due to network failures or site crashes. This is a very real problem in most networks. For example, even in the Ethernet [10], gateways can be inoperative for significant lengths of time, while the Ether segments they normally

ORIGIN POINTS & VERSION VECTORS

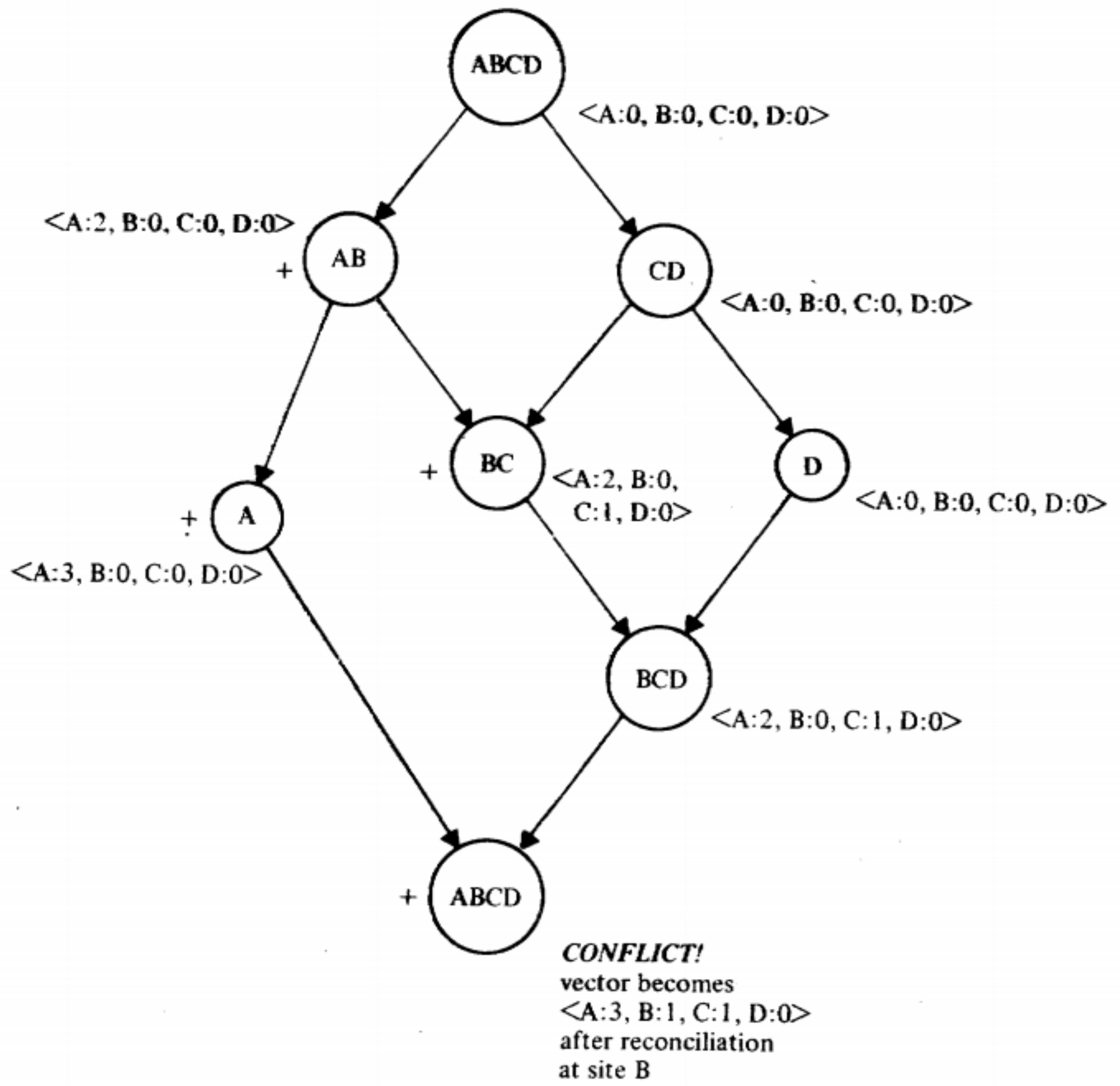


Fig. 2. Partition graph $G(f)$ for f with version vectors effective at the end of each partition.

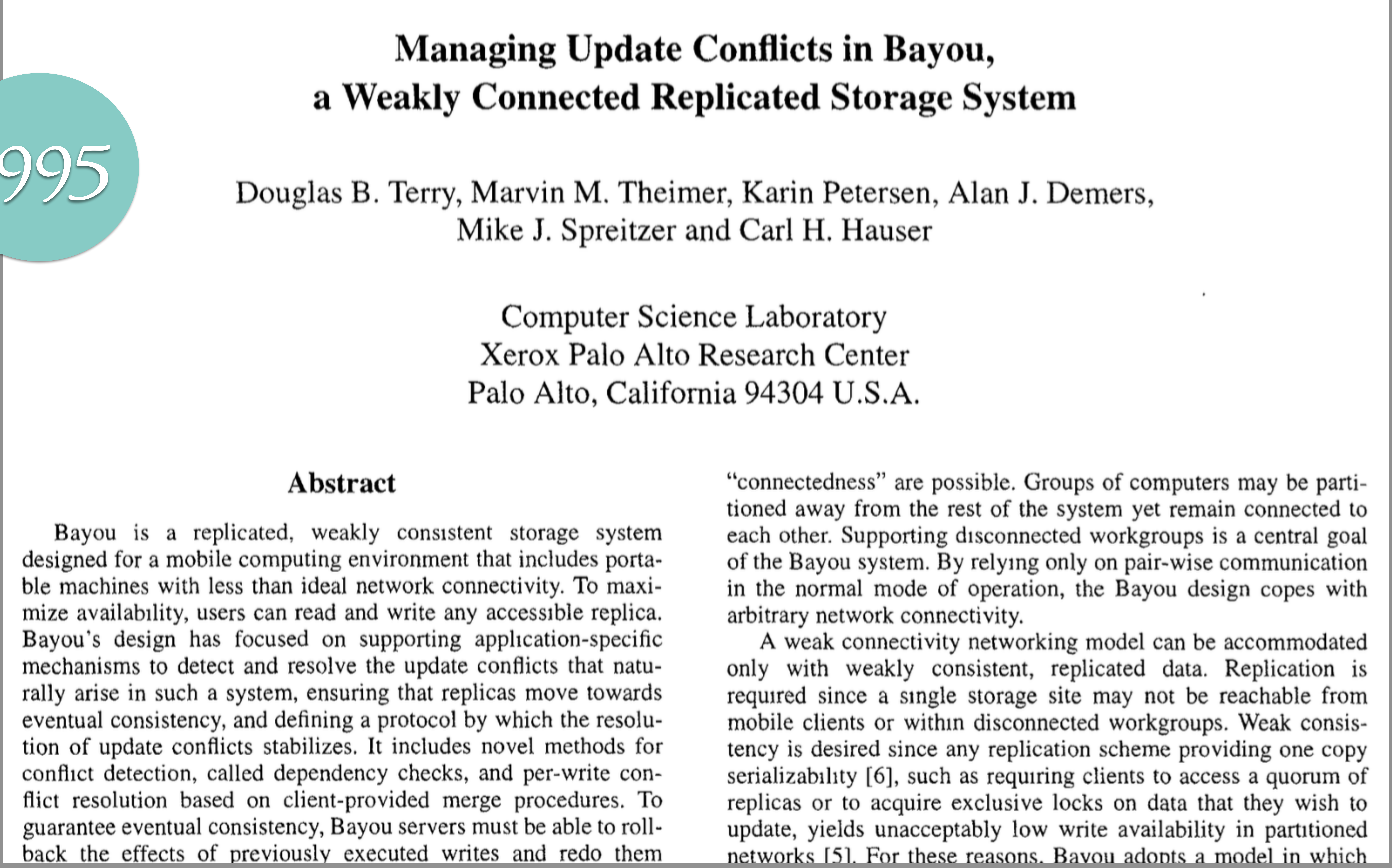
KEY *Take aways*

We need **Availability**

Gives us a mechanism for efficient conflict detection

Teaches us that networks are NOT reliable

1995



Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl H. Hauser

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.

Abstract

Bayou is a replicated, weakly consistent storage system designed for a mobile computing environment that includes portable machines with less than ideal network connectivity. To maximize availability, users can read and write any accessible replica. Bayou's design has focused on supporting application-specific mechanisms to detect and resolve the update conflicts that naturally arise in such a system, ensuring that replicas move towards eventual consistency, and defining a protocol by which the resolution of update conflicts stabilizes. It includes novel methods for conflict detection, called dependency checks, and per-write conflict resolution based on client-provided merge procedures. To guarantee eventual consistency, Bayou servers must be able to rollback the effects of previously executed writes and redo them

"connectedness" are possible. Groups of computers may be partitioned away from the rest of the system yet remain connected to each other. Supporting disconnected workgroups is a central goal of the Bayou system. By relying only on pair-wise communication in the normal mode of operation, the Bayou design copes with arbitrary network connectivity.

A weak connectivity networking model can be accommodated only with weakly consistent, replicated data. Replication is required since a single storage site may not be reachable from mobile clients or within disconnected workgroups. Weak consistency is desired since any replication scheme providing one copy serializability [6], such as requiring clients to access a quorum of replicas or to acquire exclusive locks on data that they wish to update, yields unacceptably low write availability in partitioned networks [5]. For these reasons, Bayou adopts a model in which

BAYOU *Summary*

System designed for **weak connectivity**

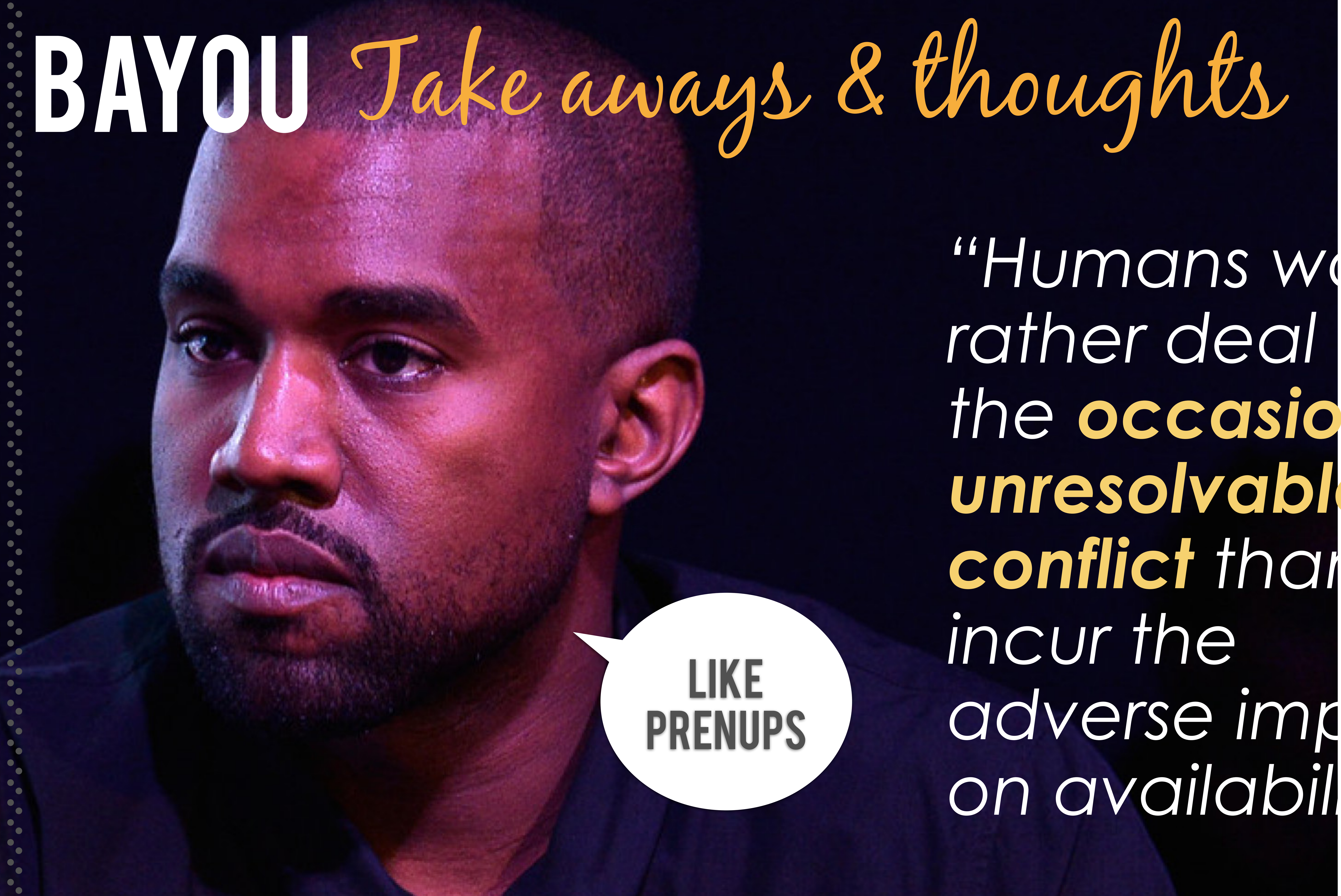
Eventual consistency via application-defined **dependency checks** and **merge procedures**

Epidemic algorithms to replicate state

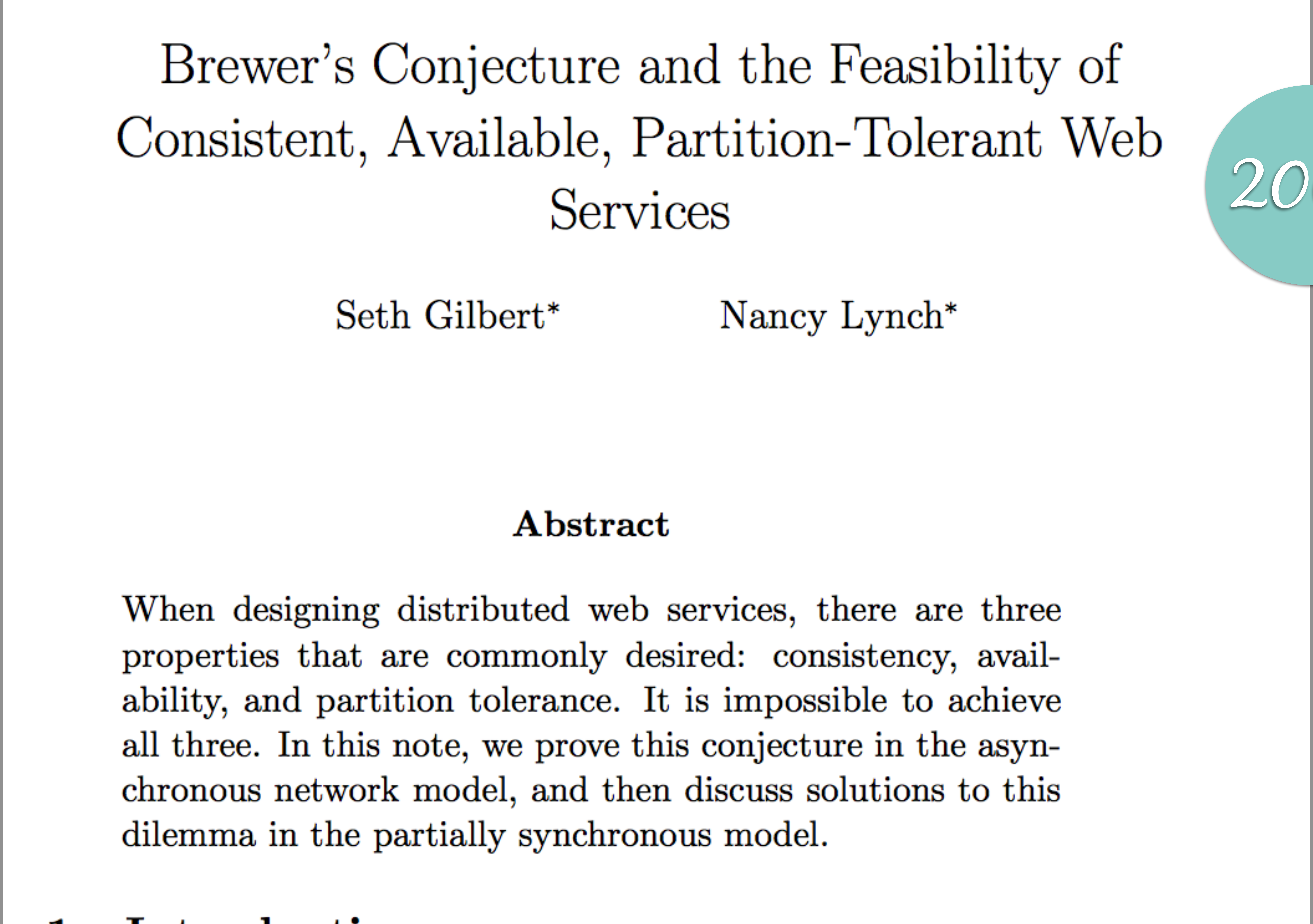
"Applications must be aware of and integrally involved in conflict detection and resolution"

Terry et. al

BAYOU *Take aways & thoughts*



"Humans wo
rather deal
the **occasional unresolvable conflict** *thar*
incur the
adverse imp
on availabil



Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

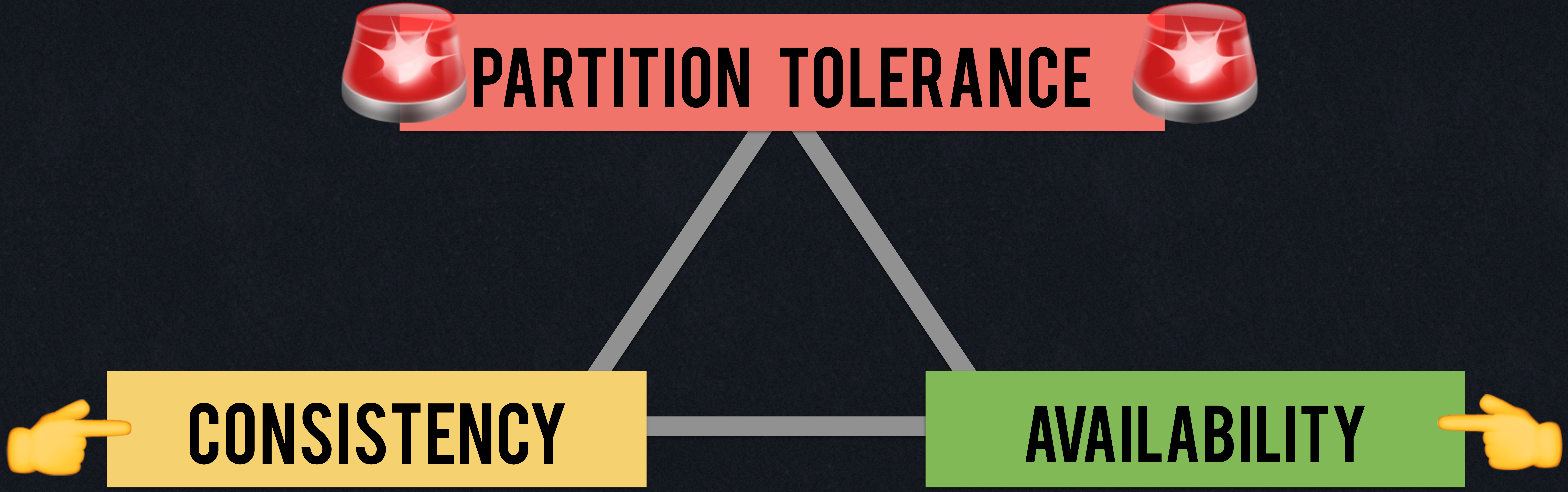
2002

Seth Gilbert* Nancy Lynch*

Abstract

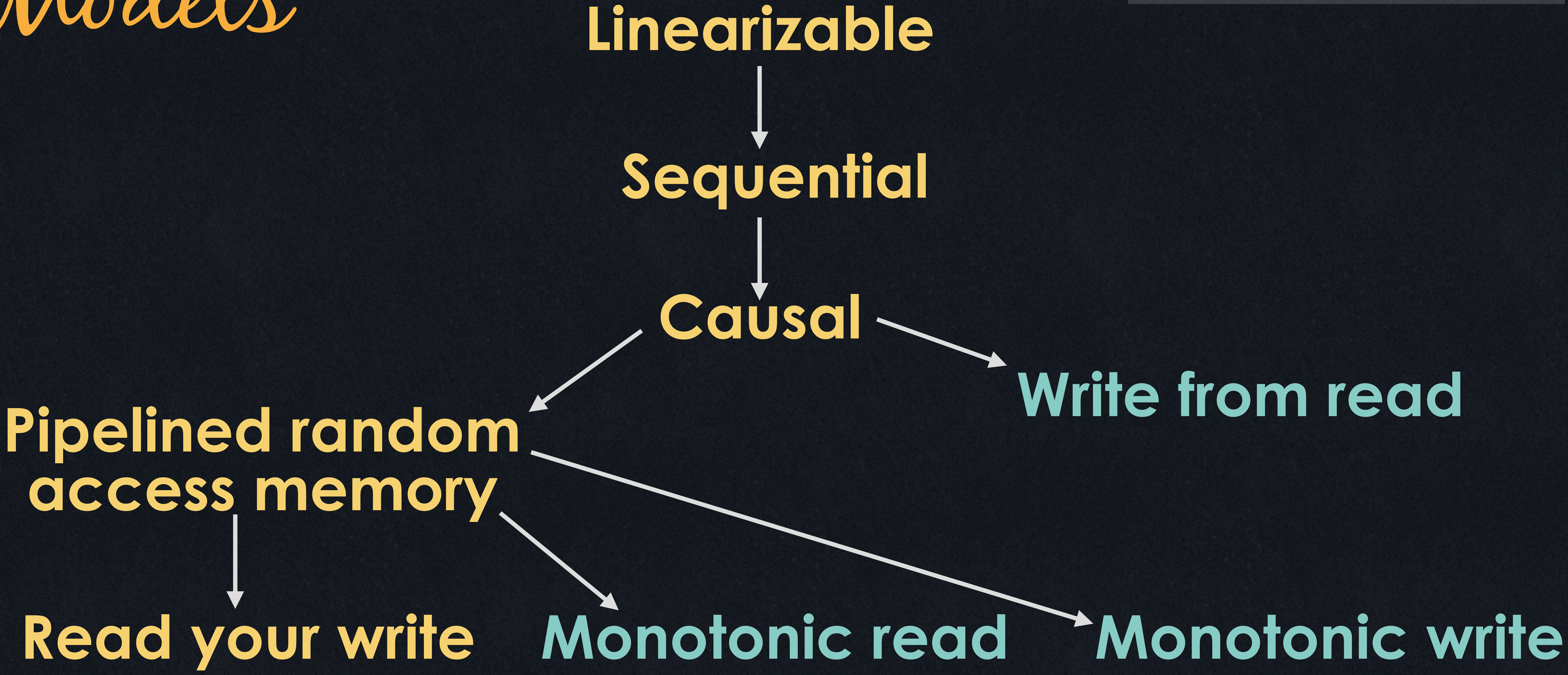
When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

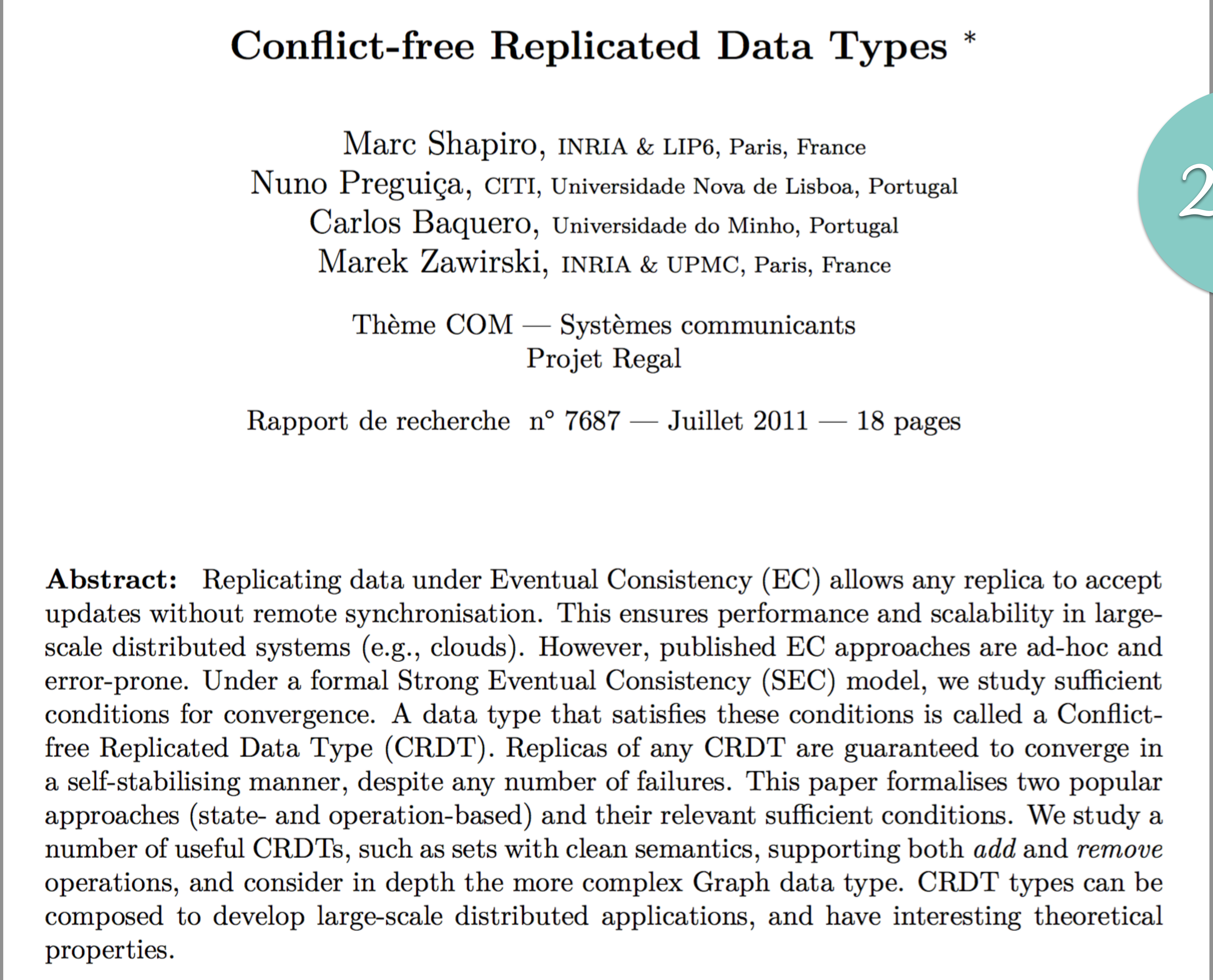
CAP *Explained*



CONSISTENCY *Models*

🟨 CP Consistency
🟦 AP Consistency





Conflict-free Replicated Data Types *

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal
Marek Zawirski, INRIA & UPMC, Paris, France

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 7687 — Juillet 2011 — 18 pages

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

2011

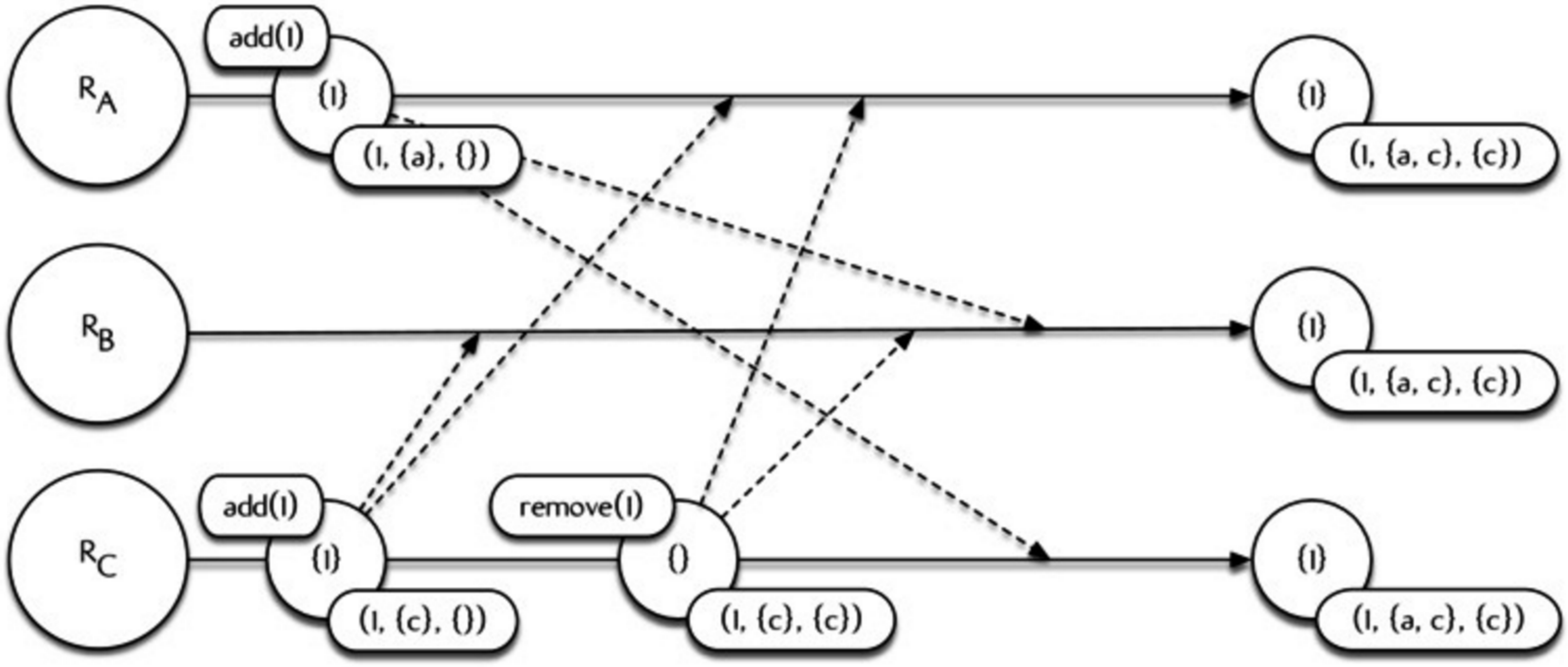
CRDTS *Summary*

Strong Eventual Consistency - apply updates immediately, no conflicts, or rollbacks

via

Mathematical properties & epidemic algorithms / gossip protocols

CRDTS *in practice*



* Stolen from Chris Meiklejohn

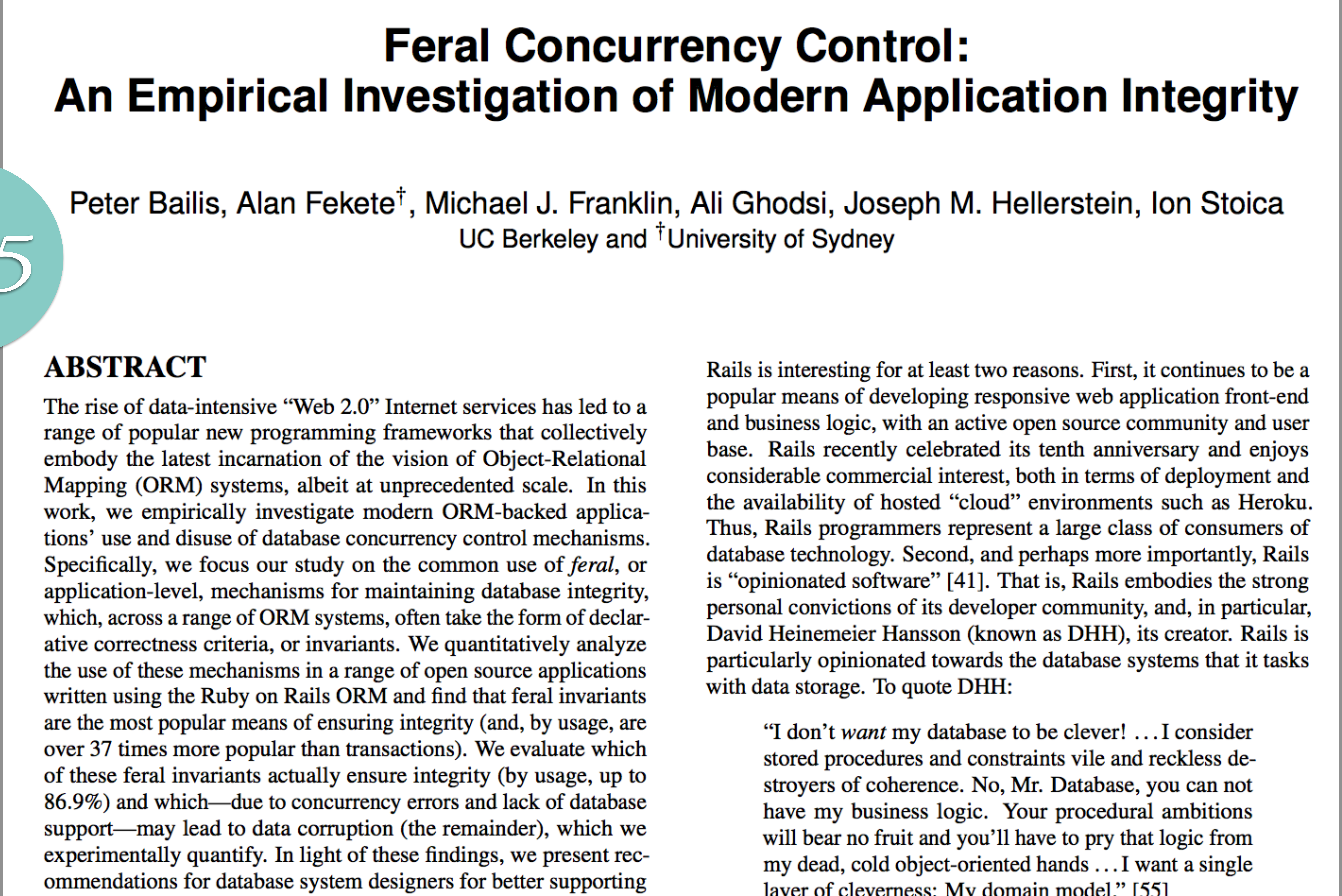
RESOLVING *Conflicts*

Applying rollbacks is hard

Restrict operation space to get provably convergent systems

Active area of research

2015



Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity

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

ABSTRACT

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

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

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

FERAL MECHANISMS *for keeping DB integrity*

Application-level mechanisms

Analyzed 67 open source Ruby on Rails Applications

Unsafe > 13% of the time
(uniqueness & foreign key constraint violations)

CONCURRENCY CONTROL *is hard!*

Availability is important to application developers

Home-rolling your own concurrency control or consensus algorithm is very hard and difficult to get correct!

CRAP! B WE STILL HAVE TO SHIP THIS SYSTEM!

SHIP THIS PILE OF BURNING TIRES? BUT HOW DO WE KNOW IF IT WORKS?
CRAP! B WE STILL HAVE TO SHIP THIS SYSTEM!

SYSTEM Verification

WHY *do we verify/test?*

We verify/test to gain **confidence** that our system is **doing the right thing** now & later

TYPES *of verification & testing*

Formal Methods

HUMAN ASSISTED PROOFS

SAFETY CRITICAL (*TLA+, COQ, ISABELLE*)

MODEL CHECKING

PROPERTIES + TRANSITIONS (*SPIN, TLA+*)

LIGHTWEIGHT FM

BEST OF BOTH WORLDS (*ALLOY, SAT*)

Testing

TOP-DOWN

FAULT INJECTORS, INPUT GENERATORS

BOTTOM-UP

LINEAGE DRIVEN FAULT INJECTORS

WHITE / BLACK BOX

WE KNOW (OR NOT) ABOUT THE SYSTEM

TYPES *of verification & testing*

Formal Methods

High investment and high reward

Considered slow & hard to use so we target small components / simplified versions of a system

Used in safety-critical domains

Testing

Pay-as-you-go & gradually increase confidence

Sacrifice rigor (less certainty) for something more reasonable

Efficacy challenged by large state space

VERIFICATION *Why so hard?*

SAFETY	LIVENESS
Nothing bad happens	Something good eventually happens
Reason about 2 system states. If steps between them preserve our invariants then we are proven safe	Reason about infinite series of system states
	Much harder to verify than safety properties

TESTING *Why so hard?*

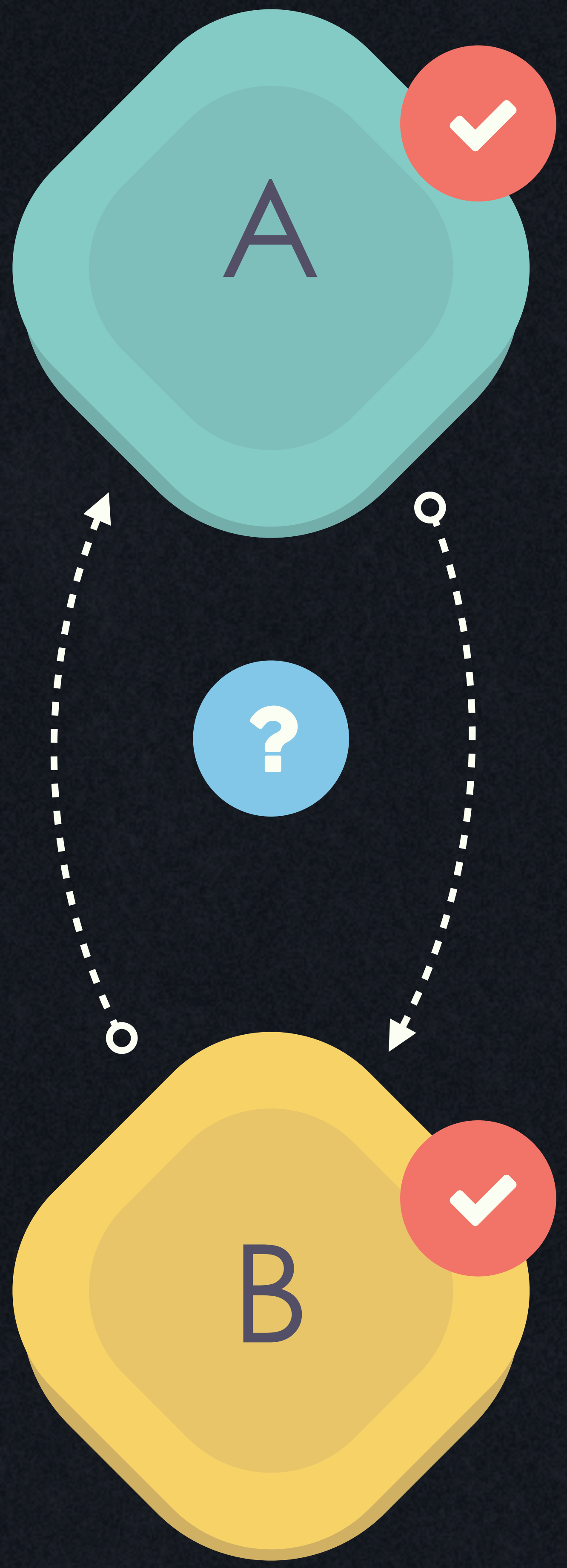
Timing & Failures

Nondeterminism

Message ordering

Concurrency

Unbounded inputs

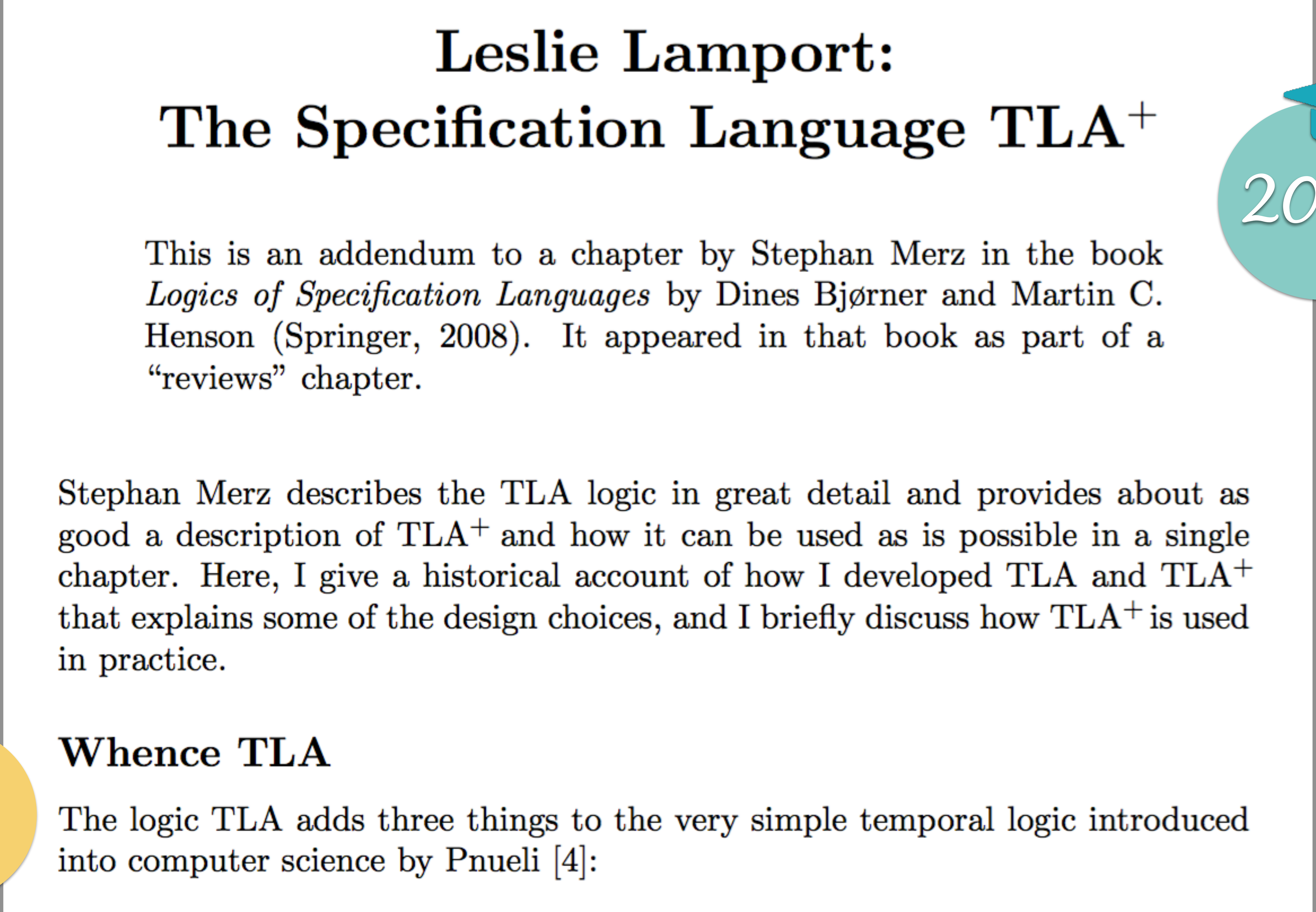


Vast state space

No centralized view

Behavior is **aggregate**

Components tested in isolation also need to be tested together



Leslie Lamport: The Specification Language TLA$^+$

2008

This is an addendum to a chapter by Stephan Merz in the book *Logics of Specification Languages* by Dines Bjørner and Martin C. Henson (Springer, 2008). It appeared in that book as part of a "reviews" chapter.

Stephan Merz describes the TLA logic in great detail and provides about as good a description of TLA$^+$ and how it can be used as is possible in a single chapter. Here, I give a historical account of how I developed TLA and TLA$^+$ that explains some of the design choices, and I briefly discuss how TLA$^+$ is used in practice.

Whence TLA

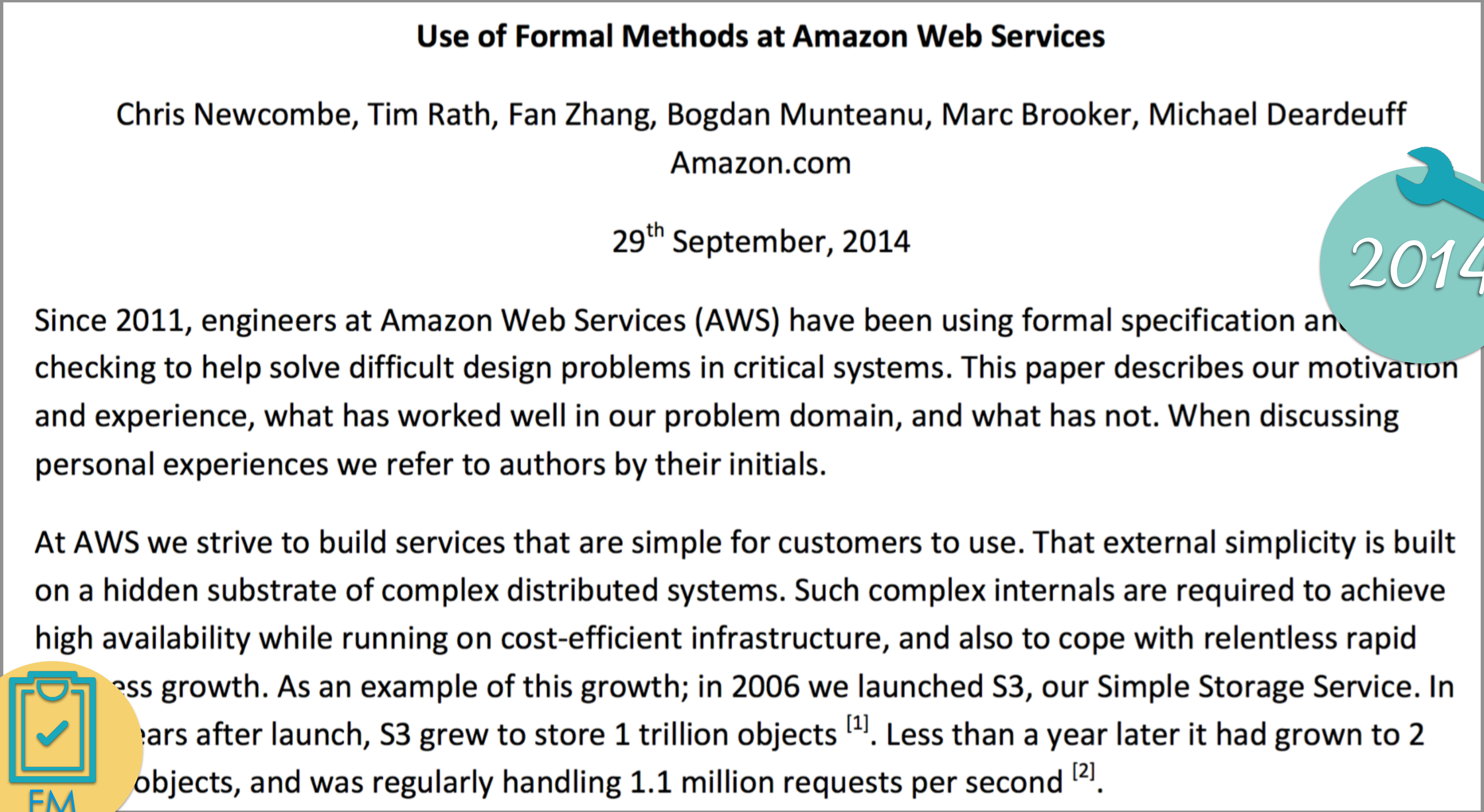
The logic TLA adds three things to the very simple temporal logic introduced into computer science by Pnueli [4]:

WHAT *is this temporal logic thing?*

TLA: is a combination of temporal logic with a logic of actions. Right logic to **express liveness properties** with predicates about a system's current & future state

TLA+: is a formal specification language used to design, model, document, and verify concurrent/ distributed systems. It verifies all traces exhaustively

One of the most commonly used Formal Methods



Use of Formal Methods at Amazon Web Services

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Amazon.com

29th September, 2014

Since 2011, engineers at Amazon Web Services (AWS) have been using formal specification an
checking to help solve difficult design problems in critical systems. This paper describes our motivation
and experience, what has worked well in our problem domain, and what has not. When discussing
personal experiences we refer to authors by their initials.

At AWS we strive to build services that are simple for customers to use. That external simplicity is built
on a hidden substrate of complex distributed systems. Such complex internals are required to achieve
high availability while running on cost-efficient infrastructure, and also to cope with relentless rapid
ess growth. As an example of this growth; in 2006 we launched S3, our Simple Storage Service. In
ears after launch, S3 grew to store 1 trillion objects [1]. Less than a year later it had grown to 2
objects, and was regularly handling 1.1 million requests per second [2].

2014

FM

TLA+ AT AMAZON *Takeaways*

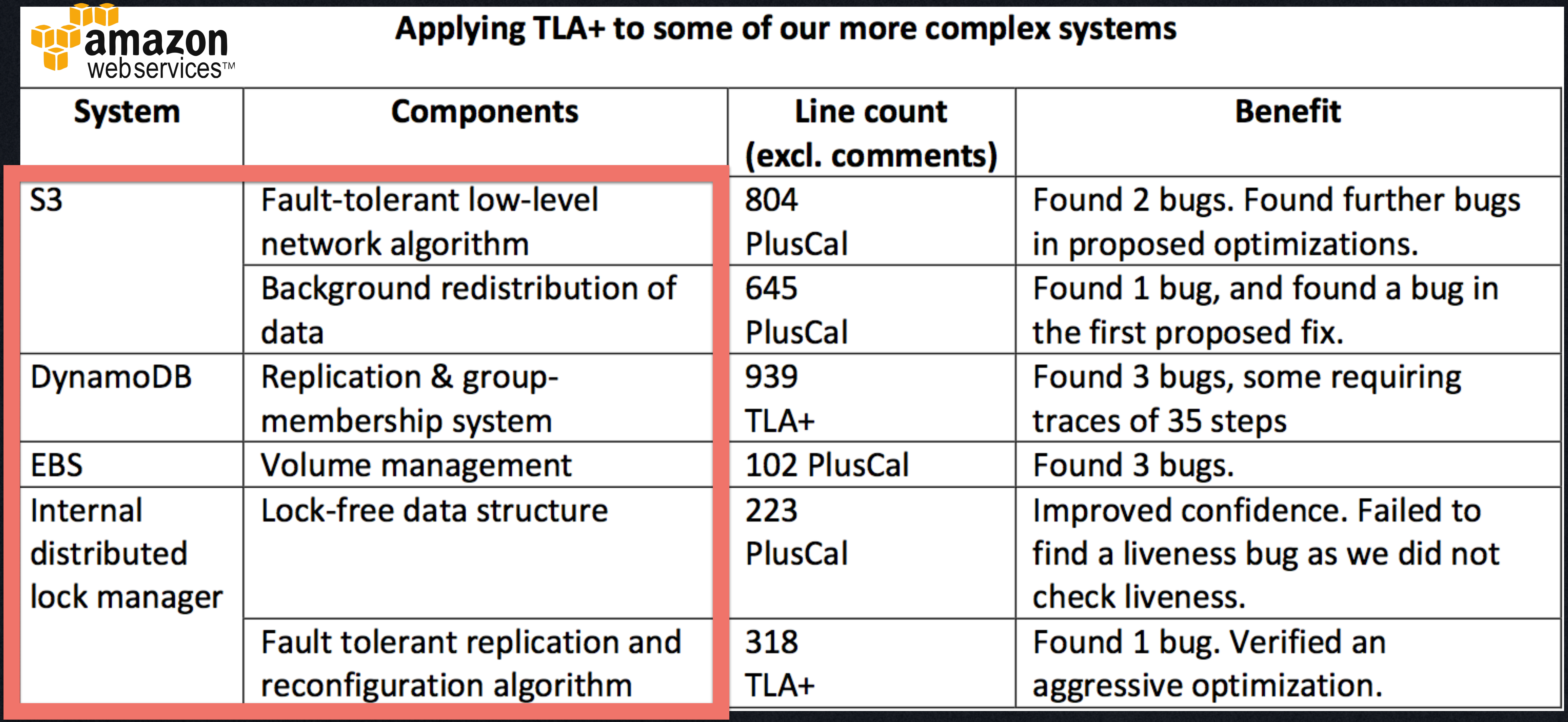
Precise specification of systems in TLA+

Used in large complex real-world systems

Found subtle bugs & FMs provided **confidence to make aggressive optimizations** w/o sacrificing system correctness

Use formal specification to teach new engineers

TLA+ AT AMAZON *Results*



amazon webservices™

Applying TLA+ to some of our more complex systems

System	Components	Line count (excl. comments)	Benefit
S3	Fault-tolerant low-level network algorithm	804 PlusCal	Found 2 bugs. Found further bugs in proposed optimizations.
	Background redistribution of data	645 PlusCal	Found 1 bug, and found a bug in the first proposed fix.
DynamoDB	Replication & group-membership system	939 TLA+	Found 3 bugs, some requiring traces of 35 steps
EBS	Volume management	102 PlusCal	Found 3 bugs.
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence. Failed to find a liveness bug as we did not check liveness.
	Fault tolerant replication and reconfiguration algorithm	318 TLA+	Found 1 bug. Verified an aggressive optimization.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, *University of Toronto*

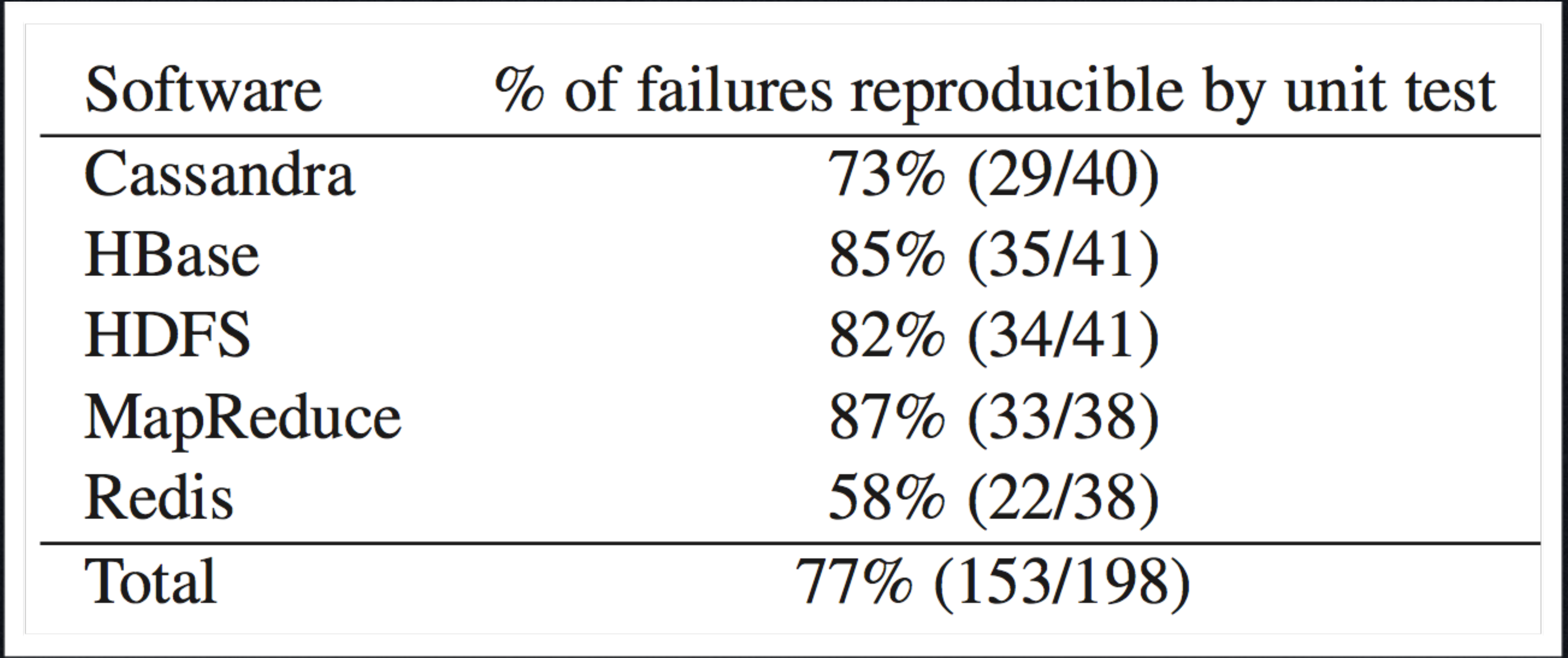
https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

2014

TEST

KEY *Takeaways*

Used error logs to diagnose & reproduce failures



Software	% of failures reproducible by unit test
Cassandra	73% (29/40)
HBase	85% (35/41)
HDFS	82% (34/41)
MapReduce	87% (33/38)
Redis	58% (22/38)
Total	77% (153/198)

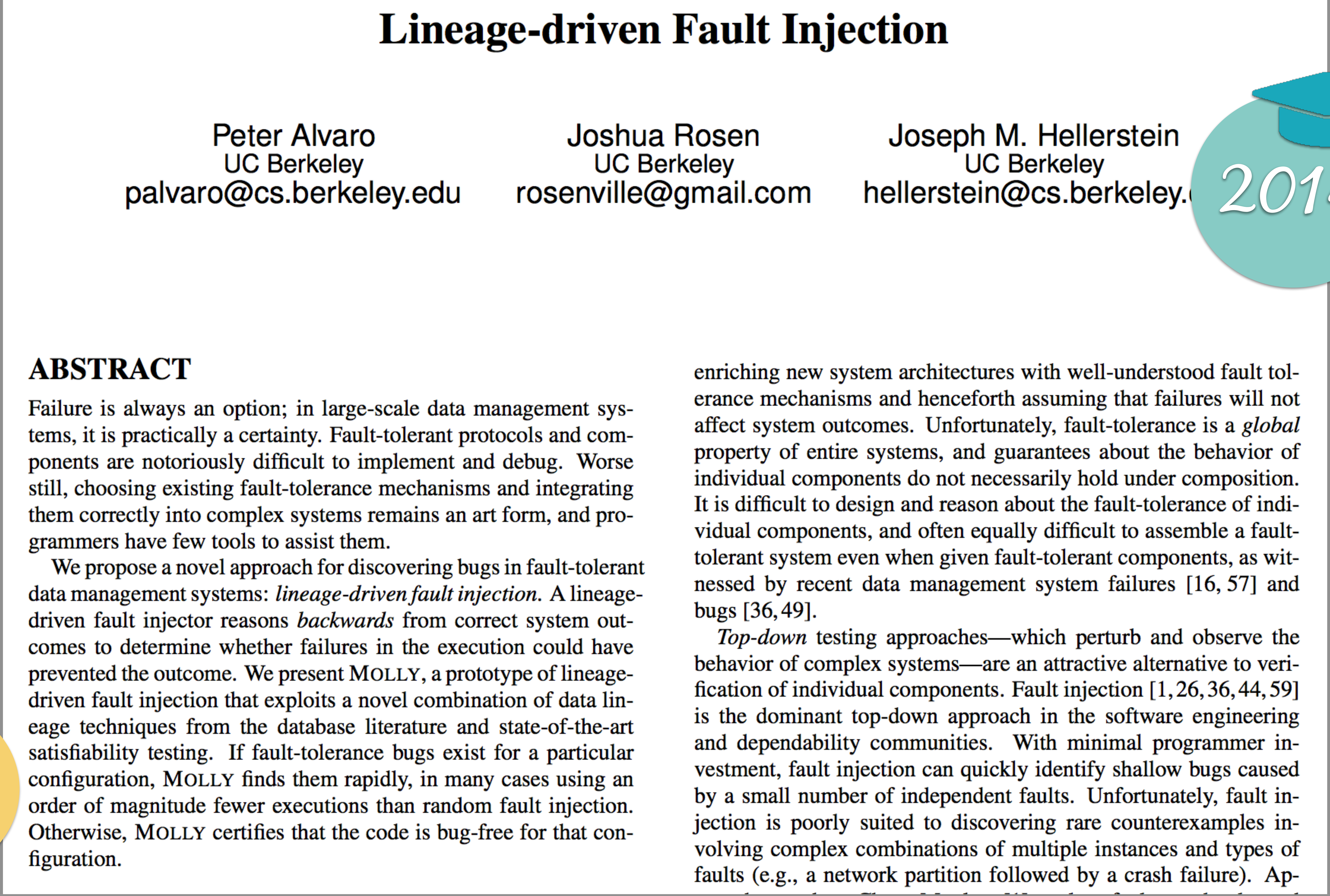
Aspirator (their static checker) found 121 new bugs & 379 bad practices!

Failures require **only 3 nodes to reproduce**. Multiple inputs needed (~ 3) in the correct order

Complex sequences of events but **74% errors found are deterministic**

77% failures can be reproduced by a unit test

Faulty error handling code culprit



Lineage-driven Fault Injection

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joshua Rosen
UC Berkeley
rosenville@gmail.com

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

2014

ABSTRACT

Failure is always an option; in large-scale data management systems, it is practically a certainty. Fault-tolerant protocols and components are notoriously difficult to implement and debug. Worse still, choosing existing fault-tolerance mechanisms and integrating them correctly into complex systems remains an art form, and programmers have few tools to assist them.

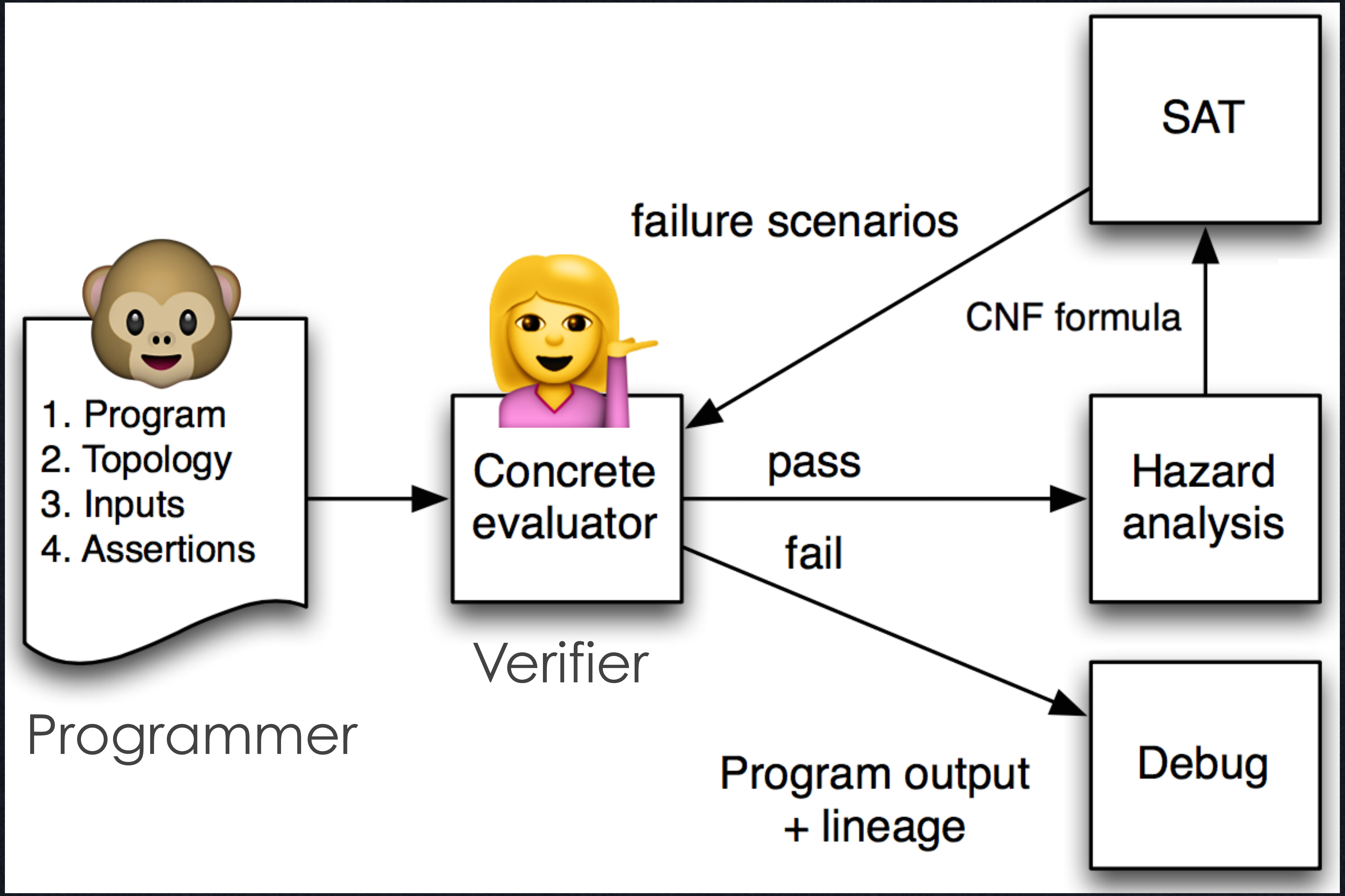
We propose a novel approach for discovering bugs in fault-tolerant data management systems: *lineage-driven fault injection*. A lineage-driven fault injector reasons *backwards* from correct system outcomes to determine whether failures in the execution could have prevented the outcome. We present MOLLY, a prototype of lineage-driven fault injection that exploits a novel combination of data lineage techniques from the database literature and state-of-the-art satisfiability testing. If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly, in many cases using an order of magnitude fewer executions than random fault injection. Otherwise, MOLLY certifies that the code is bug-free for that configuration.

enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a *global* property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [16, 57] and bugs [36, 49].

Top-down testing approaches—which perturb and observe the behavior of complex systems—are an attractive alternative to verification of individual components. Fault injection [1, 26, 36, 44, 59] is the dominant top-down approach in the software engineering and dependability communities. With minimal programmer investment, fault injection can quickly identify shallow bugs caused by a small number of independent faults. Unfortunately, fault injection is poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of faults (e.g., a network partition followed by a crash failure). Ap-

TEST

MOLLY *Highlights*

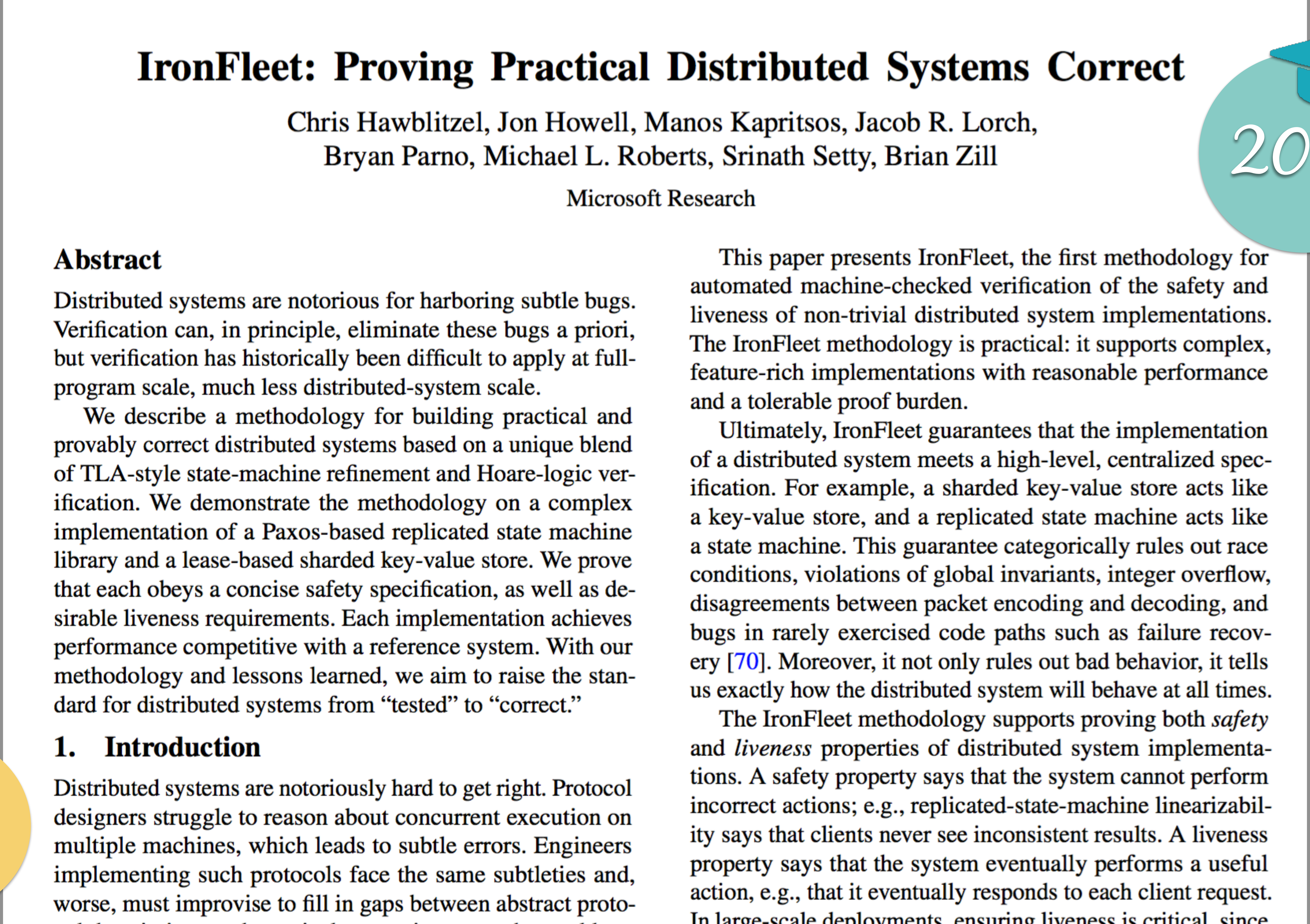


MOLLY runs and observes execution, & picks a fault for the next execution. Program is ran again and results are observed

Reasons backwards from correct system outcomes & determines if a failure could have prevented it

Molly **only injects the failures it can prove might affect an outcome**

"Presents a **middle ground** between **pragmatism** and **formalism**, dictated by the importance of verifying fault tolerance in spite of the complexity of the space of faults"



IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

2015

Abstract

Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs a priori, but verification has historically been difficult to apply at full-program scale, much less distributed-system scale.

We describe a methodology for building practical and provably correct distributed systems based on a unique blend of TLA-style state-machine refinement and Hoare-logic verification. We demonstrate the methodology on a complex implementation of a Paxos-based replicated state machine library and a lease-based sharded key-value store. We prove that each obeys a concise safety specification, as well as desirable liveness requirements. Each implementation achieves performance competitive with a reference system. With our methodology and lessons learned, we aim to raise the standard for distributed systems from "tested" to "correct."

1. Introduction

Distributed systems are notoriously hard to get right. Protocol designers struggle to reason about concurrent execution on multiple machines, which leads to subtle errors. Engineers implementing such protocols face the same subtleties and, worse, must improvise to fill in gaps between abstract proto-

This paper presents IronFleet, the first methodology for automated machine-checked verification of the safety and liveness of non-trivial distributed system implementations. The IronFleet methodology is practical: it supports complex, feature-rich implementations with reasonable performance and a tolerable proof burden.

Ultimately, IronFleet guarantees that the implementation of a distributed system meets a high-level, centralized specification. For example, a sharded key-value store acts like a key-value store, and a replicated state machine acts like a state machine. This guarantee categorically rules out race conditions, violations of global invariants, integer overflow, disagreements between packet encoding and decoding, and bugs in rarely exercised code paths such as failure recovery [70]. Moreover, it not only rules out bad behavior, it tells us exactly how the distributed system will behave at all times.

The IronFleet methodology supports proving both *safety* and *liveness* properties of distributed system implementations. A safety property says that the system cannot perform incorrect actions; e.g., replicated-state-machine linearizability says that clients never see inconsistent results. A liveness property says that the system eventually performs a useful action, e.g., that it eventually responds to each client request. In large-scale deployments, ensuring liveness is critical, since

FM

IRONFLEET *Takeaways*

First automated machine-checked verification of safety and **liveness** of a non-trivial distributed system implementation

Guarantees a system implementation meets a high-level specification

Rules out race conditions,…, invariant violations, & bugs!

Uses **TLA style state-machine refinements** to reason about **protocol level concurrency** (ignoring implementation)

plus

Floyd-Hoare style **imperative verification** to reason about **implementation** complexities (ignoring concurrency)

KEY *Takeaways*

High-level spec (§3.1)
refinement (§3.3)
Distributed protocol (§3.2)
refinement (§3.5)
Implementation (§3.4)
H0
H1
H2
H3
H4
P0
P1
P2
P3
I0
I1
I2
I3

"… As the developer writes a given method or proof, she typically sees **feedback in 1–10 seconds indicating whether the verifier is satisfied**. Our build system tracks dependencies across files and outsources, in parallel, each file's verification to a cloud virtual machine. While a full integration build done serially requires approximately 6 hours, in practice, the **developer rarely waits more than 6–8 minutes**"

KEEP *In Mind*

Formally specified algorithms gives us the **most confidence** that our systems are doing the right thing

No testing strategy will ever give you a completeness guarantee that no bugs exist

HEY BRITNEY, I'M READY TO BUILD BETTER SOFTWARE
AND TEST IT TOO JUSTIN!

TL;DR *Consistency*

We want highly available systems so we must use weaker forms of consistency (remember CAP)

Application semantics helps us make better tradeoffs

Do not recreate the wheel, leverage existing research allows us to not repeat past mistakes

Forced into a feral world but this may change soon!

TL;DR *Verification*

Verification of distributed systems is a complicated matter but we still need it

Today we leverage a multitude of methods to gain confidence that we are doing the right thing

Formal vs testing lines are starting to get blurry

Still not as many tools as we should have. **We wish for more confidence with less work**

Thank you!

GITHUB.COM/RANDOMMOOD/QCONSF2015

@Caitie - @Randommood