We hear you like PAPERS
CAITIE MCCAFFREY

@Caitie
DISTRIBUTED Systems
WE ARE ALL BUILDING DISTRIBUTED SYSTEMS
TWITTER SERVICES
ACADEMIC Papers
EVENTUAL Consistency
THINKING Consistency

1983
Detection of Mutual Inconsistency in Distributed Systems

1995
Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System

2002
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 Now
Detection of Mutual Inconsistency in Distributed Systems

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.
“In some environments it is desirable or necessary to permit users to continue modifying resources such as files when the network is partitioned.”
Fig. 2. Partition graph $G(f)$ for $f$ with version vectors effective at the end of each partition.
Version Vectors

“A Version Vector for a file $f$ is a sequence of $n$ pairs, where $n$ is the number of sites at which $f$ is stored ... the $i$th vector entry counts the number $V_i$ of updates to $f$ made at site $S_i$”

$$\langle A:9, B:7, C:22, D:3 \rangle$$
Compatible Version Vectors

\[ \langle A:1, B:2, C:4, D:3 \rangle \]

\[ \langle A:0, B:2, C:2, D:3 \rangle \]
Incompatible Version Vectors

\langle A:1, B:2, C:4, D:3 \rangle

\langle A:1, B:2, C:3, D:4 \rangle
Partition Graph

ABCD <A:0, B:0, C:0, D:0>
Partition Graph

ABCD

< A:0, B:0, C:0, D:0 >

AB

< A:0, B:0, C:0, D:0 >

CD

< A:0, B:0, C:0, D:0 >
Partition Graph

ABCD

AB

CD
Partition Graph

ABCD

A
B
C
D

T_2
No Conflict!

Partition Graph

\[
\begin{align*}
&A: 1, B: 0, C: 0, D: 0 & \quad AB \\
&A: 0, B: 0, C: 0, D: 0 & \quad CD \\
&A: 1, B: 0, C: 0, D: 0 & \quad D
\end{align*}
\]
No Conflict!

Partition Graph

\[ \langle A:0, B:0, C:0, D:0 \rangle \]

\[ \langle A:1, B:0, C:0, D:0 \rangle \]

\[ \langle A:2, B:0, C:0, D:0 \rangle \]

\[ \langle A:1, B:0, C:1, D:0 \rangle \]

\[ \langle A:0, B:0, C:0, D:0 \rangle \]
Partition Graph

T_2
Partition Graph

Conflict!

<\(A:0, B:0, C:0, D:0\)>

<\(A:1, B:0, C:0, D:0\)>

<\(A:2, B:0, C:0, D:0\)>

<\(A:1, B:0, C:1, D:0\)>

<\(A:0, B:0, C:0, D:0\)>

<\(A:0, B:0, C:0, D:0\)>

<\(A:0, B:0, C:0, D:0\)>

<\(A:0, B:0, C:0, D:0\)>

<\(A:0, B:0, C:0, D:0\)>

<\(A:0, B:0, C:0, D:0\)>

<\(A:0, B:0, C:0, D:0\)>

<\(A:0, B:0, C:0, D:0\)>
Conflict Resolution

“A conflict detection mechanism, while valuable, has increased effect if there is also a method for reconciling conflicts automatically”
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 developer defined merge procedures

Epidemic algorithms to replicate state
“Applications must be aware of and integrally involved in conflict detection and resolution”

Terry et. al
Humans would rather deal with the occasional unresolvable conflict than incur the adverse impact on availability.
Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

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.

1. Introduction
CAP Theorem

- Partition Tolerance
- Consistency
- Availability
**CONSISTENCY Models**

- Linearizable
  - Sequential
    - Causal
      - Write from read
      - Monotonic read
      - Monotonic write
  - Pipelined random access memory
    - Read your write

- 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.
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
Applying rollbacks is hard

Restrict operation space to get provably convergent systems

Active area of research
RESOLVING Conflicts

Applying rollbacks is hard

Restrict operation space to get provably convergent systems

Active area of research
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!
TL;DR Eventual 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!
Thank you!
Thank you!

Follow your dreams!