Papers We Love SF: Bayou

Peter Bailis @pbailis
UC Berkeley
17 December 2014

Who am I?

Ph.D. candidate, 4th year student at Berkeley

Study high performance distributed databases

» When do we need coordination?

» What happens if we don't coordinate?

Graduating 2015!

http://bailis.org/

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 roll-back the effects of previously executed writes and redo them according to a global serialization order. Furthermore, Bayou permits clients to observe the results of all writes received by a server, including tentative writes whose conflicts have not been ultimately resolved. This paper presents the motivation for and design of these mechanisms and describes the experiences gained with an initial implementation of the system.

1. Introduction

The Bayou storage system provides an infrastructure for collaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. The advent of mobile computers, in the form of laptops and personal digital assistants (PDAs) enables the use of computational facilities away from the usual "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 clients can read and write to any replica without the need for explicit coordination with other replicas. Every computer eventually receives updates from every other, either directly or indirectly, through a chain of pair-wise interactions.

Unlike many previous systems [12, 27], our goal in designing the Bayou system was *not* to provide transparent replicated data support for existing file system and database applications. We believe that applications must be aware that they may read weakly consistent data and also that their write operations may conflict with those of other users and applications. Moreover, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application.

To this end, Bayou provides system support for application-specific conflict detection and resolution. Previous systems, such as Locus [30] and Coda [17], have proven the value of semantic

Why Bayou?

"NoSQL" before "NoSQL"

Learn from our predecessors

Rethink application, system boundaries

Lucid discussion of systems challenges

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

NEWTON
TECHNOLOGY

10:45 Mon 12/30
All Notes
Sat 12/18

Hello, iPhone

New

Newton 10:44 368K

MessagePad 2100

AT&T 1:05 PM
Notes
Hello, Ne...
Today Jul 7 12:53 PM

Hello, Newton.

Bayou Goals

Storage system for mobile devices:

Handle frequent disconnections

Network Working Group
RFC # 677
NIC # 31507

Paul R. Johnson (BBN-TENEX)
Robert H. Thomas (BBN-TENEX)
January 27, 1975

The Maintenance of Duplicate Databases

Preface:

This RFC is a working paper on the problem of maintaining duplicated databases in an ARPA-like network. It briefly discusses the general duplicate database problem, and then outlines in some detail a solution for a particular type of duplicate database. The concepts developed here were used in the design of the User Identification Database for the TIP user authentication and accounting system. We believe that these concepts are generally applicable to distributed database problems.

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 roll-back the effects of previously executed writes and redo them according to a global serialization order. Furthermore, Bayou permits clients to observe the results of all writes received by a server, including tentative writes whose conflicts have not been ultimately resolved. This paper presents the motivation for and design of these mechanisms and describes the experiences gained with an initial implementation of the system.

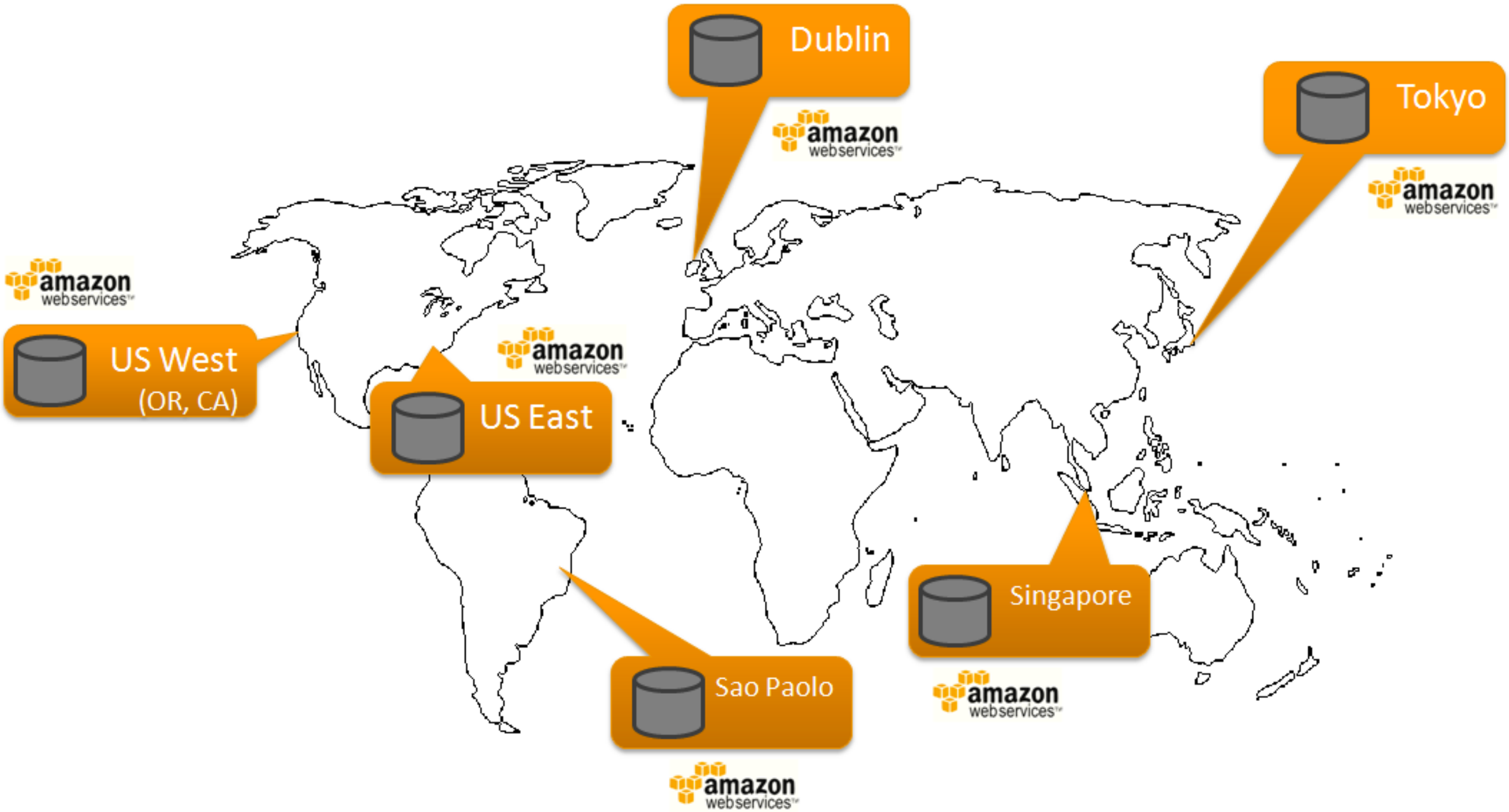
1. Introduction

The Bayou storage system provides an infrastructure for collaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. The advent of mobile computers, in the form of laptops and personal digital assistants (PDAs) enables the use of computational facilities away from the usual "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 clients can read and write to any replica without the need for explicit coordination with other replicas. Every computer eventually receives updates from every other, either directly or indirectly, through a chain of pair-wise interactions.

Unlike many previous systems [12, 27], our goal in designing the Bayou system was *not* to provide transparent replicated data support for existing file system and database applications. We believe that applications must be aware that they may read weakly consistent data and also that their write operations may conflict with those of other users and applications. Moreover, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application.

To this end, Bayou provides system support for application-specific conflict detection and resolution. Previous systems, such as Locus [30] and Coda [17], have proven the value of semantic


Dublin
amazon web services
Tokyo
amazon web services
amazon web services
US West
(OR, CA)
amazon web services
US East
Singapore
amazon web services
Sao Paolo
amazon web services

Bayou Goals

Storage system for mobile devices:

Handle frequent disconnections

Reason about distribution explicitly

A Note on Distributed Computing

Jim Waldo
Geoff Wyant
Ann Wollrath
Sam Kendall

SMLI TR-94-29 November 1994

Abstract:

We argue that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space. These differences are required because distributed systems require that the programmer be aware of latency, have a different model of memory access, and take into account issues of concurrency and partial failure.

We look at a number of distributed systems that have attempted to paper over the distinction between local and remote objects, and show that such systems fail to support basic requirements of robustness and reliability. These failures have been masked in the past by the small size of the distributed systems that have been built. In the enterprise-wide distributed systems foreseen in the near future, however, such a masking will be impossible.

We conclude by discussing what is required of both systems-level and application-level programmers and designers if one is to take distribution seriously.

Bayou Goals

Storage system for mobile devices:

Handle frequent disconnections

Reason about distribution explicitly

Facilitate conflict detection and resolution

How do we build a system that allows update-anywhere but allows users to easily build correct applications?

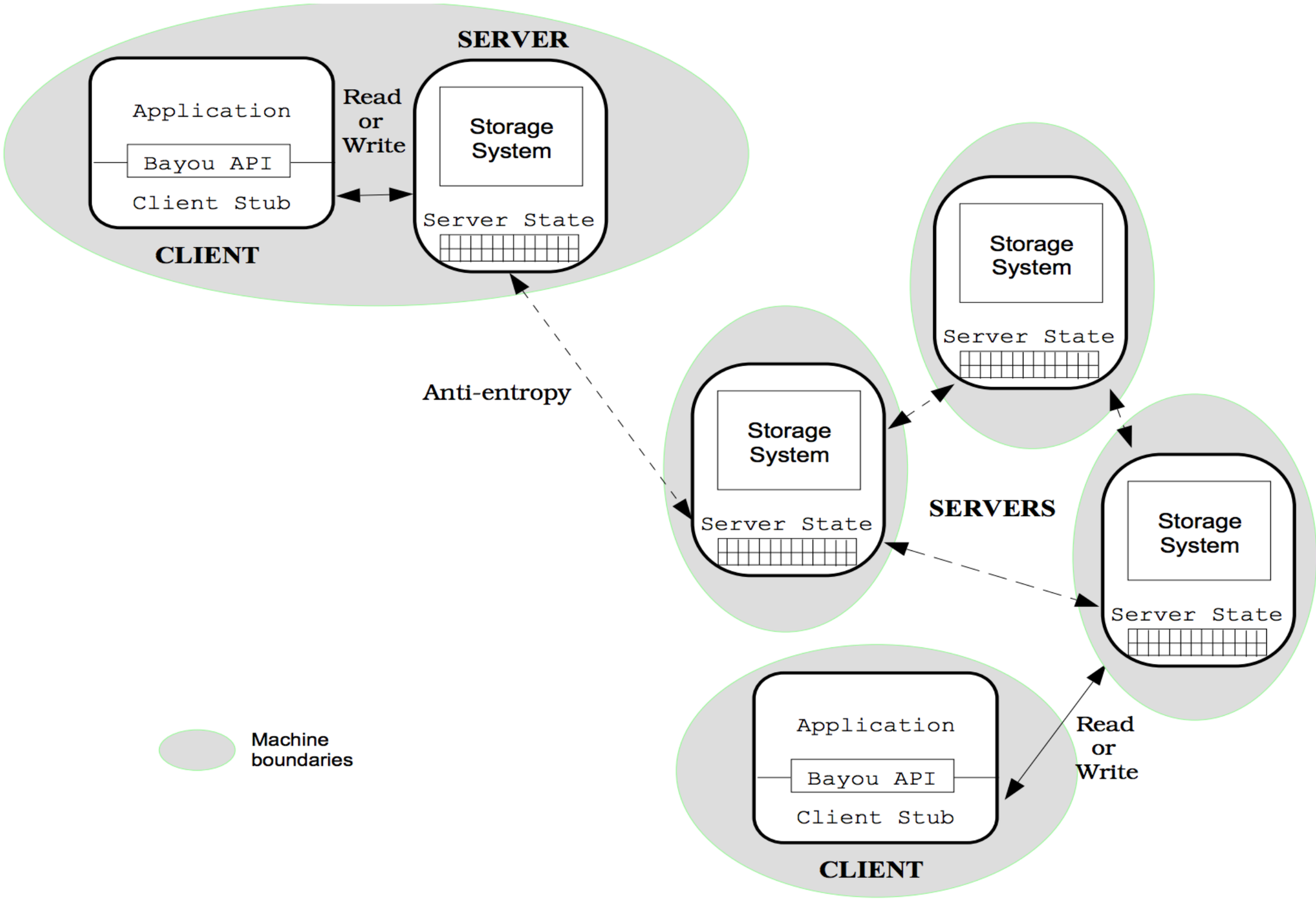


Figure 1. Bayou System Model

Basic Architecture

Optimistically replicated distributed database

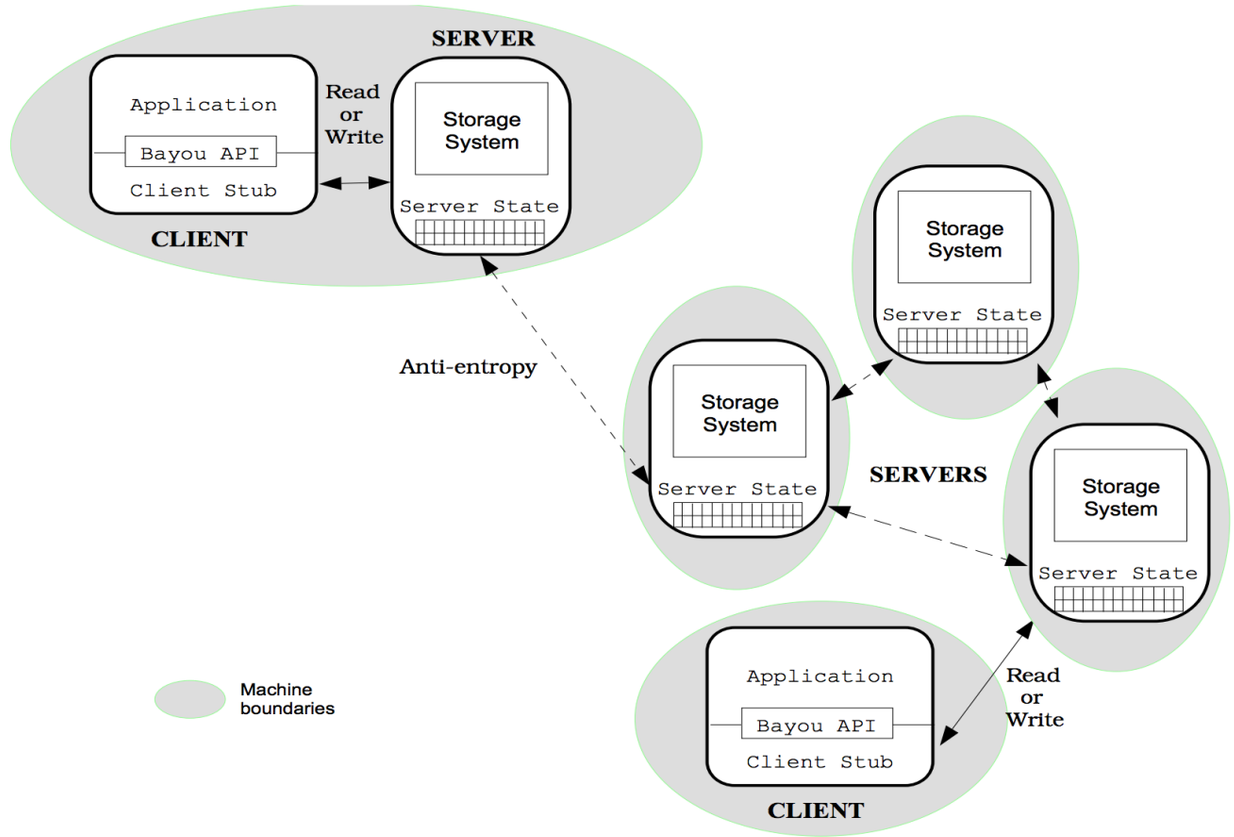


Figure 1. Bayou System Model

Optimistic replication

Yasushi Saito
Hewlett-Packard Laboratories, Palo Alto, CA (USA)
and
Marc Shapiro
Microsoft Research Ltd., Cambridge (UK)

Data replication is a key technology in distributed data sharing systems, enabling higher availability and performance. This paper surveys optimistic replication algorithms that allow replica contents to diverge in the short term, in order to support concurrent work practices and to tolerate failures in low-quality communication links. The importance of such techniques is increasing as collaboration through wide-area and mobile networks becomes popular.

Optimistic replication techniques are different from traditional "pessimistic" ones. Instead of synchronous replica coordination, an optimistic algorithm propagates changes in the background, discovers conflicts after they happen and reaches agreement on the final contents incrementally.

We explore the solution space for optimistic replication algorithms. This paper identifies key challenges facing optimistic replication systems — ordering operations, detecting and resolving conflicts, propagating changes efficiently, and bounding replica divergence — and provides a comprehensive survey of techniques developed for addressing these challenges.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Replication, Distributed Systems, Internet

---

1. INTRODUCTION

Data replication consists of maintaining multiple copies of critical data, called *replicas*, on separate computers. It is a critical enabling technology of distributed services, improving both their availability and performance. Availability is improved by allowing access to the data even when some of the replicas are unavailable. Performance improvements concern reduced latency, which improves by letting users access nearby replicas and avoiding re-

Basic Architecture

Optimistically replicated distributed database

Update-anywhere; CAP "AP"

» Guarantees availability

» Low latency and scalability!

Use anti-entropy to exchange updates

» Pick your favorite

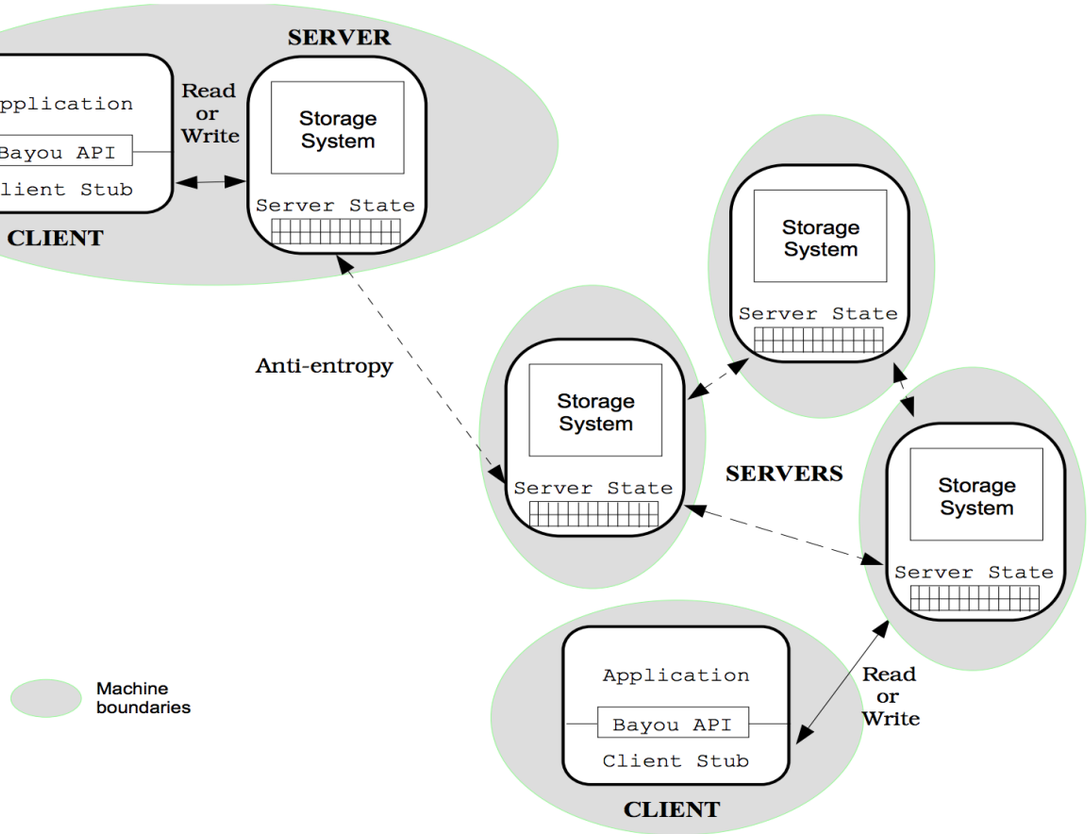
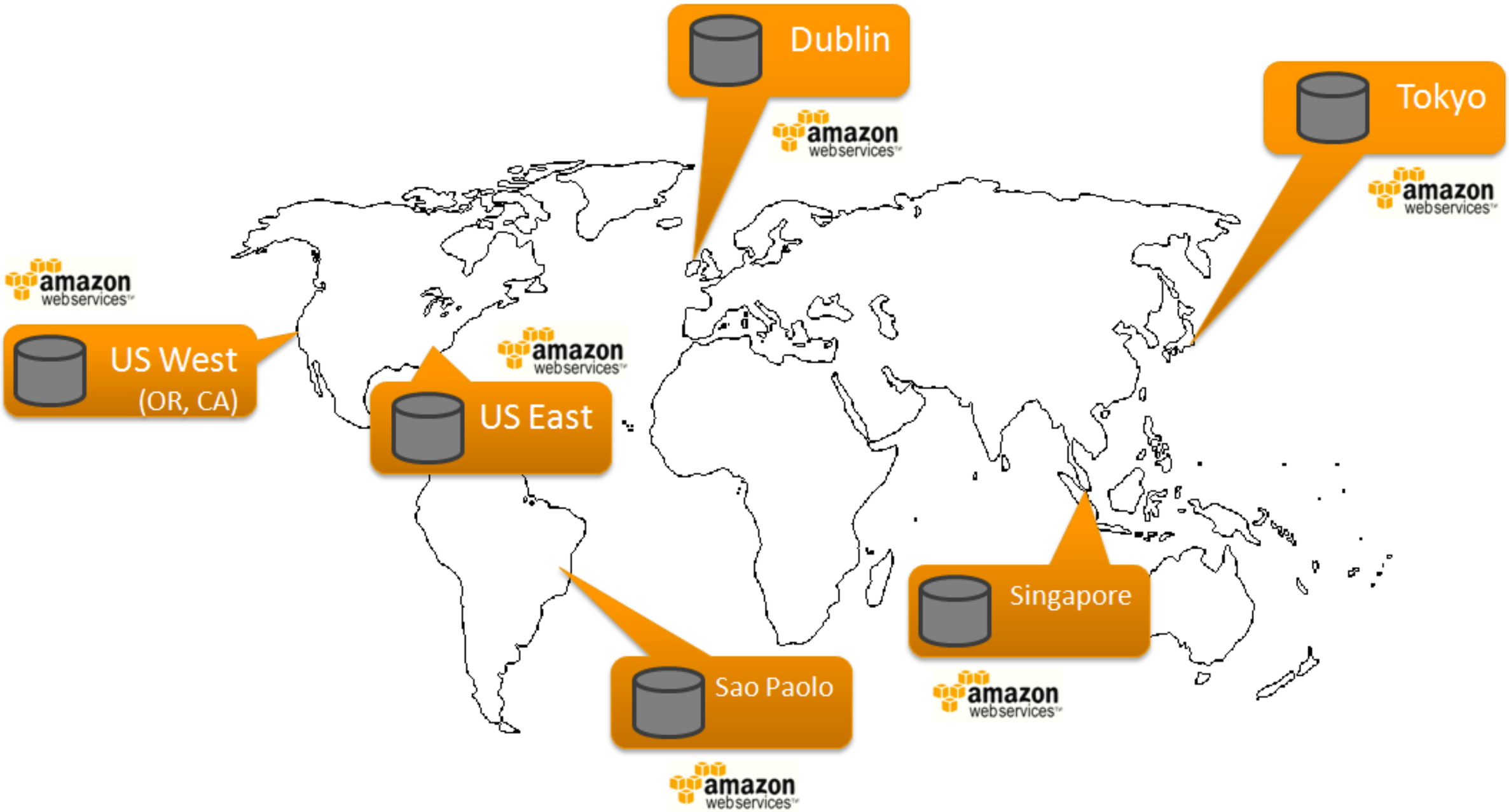


Figure 1. Bayou System Model


Dublin
amazon web services
Tokyo
amazon web services
amazon web services
US West
(OR, CA)
amazon web services
US East
Singapore
amazon web services
Sao Paolo
amazon web services

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

Main problem: “conflicts”

What happens if two clients simultaneously update the same piece of data?

Example:

Peter wants to book Fastly HQ on 12/17 at 7PM
Ines wants to book Fastly HQ on 12/17 at 7PM
Peter and Ines are connected to different servers
What will happen?

Example:

Mike wants to give Peter $100
Carol wants to give Peter $100
Mike and Carol are connected to different servers
What will happen?

Main problem: “conflicts”

What happens if two clients simultaneously update the same piece of data?

It depends on the application!

Traditional DB (“serializability”): always* matters!

Bayou: let the application help us out!

Main problem: “conflicts”

Bayou writes have three components:

» An update function: the actual write

» A dependency check: conflict detection

» A merge function: conflict compensation

> If **dependency check** passes**:**
>
> apply **update function**
>
> Else:
>
> apply **merge function**

Example:

Booking tonight's meetup:

» Update function: insert 12/17/14, "Fastly HQ", Ines

» Dependency check: Is the requested time and location available?

» Merge function: try to find another time based on provided alternates

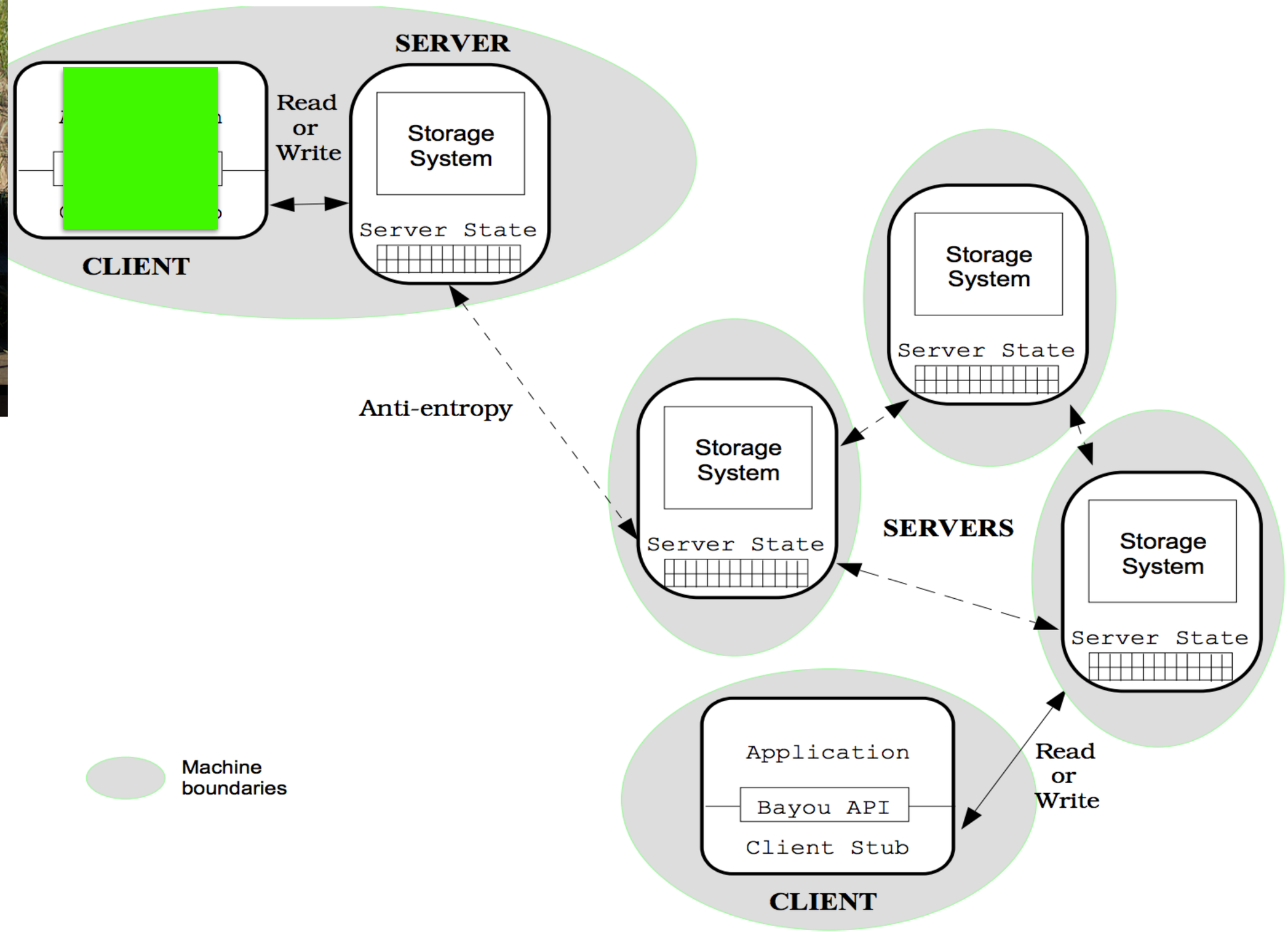
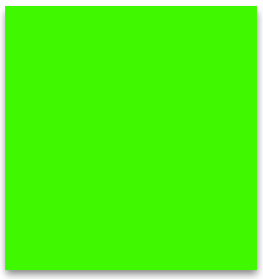


Figure 1. Bayou System Model

Inside the server:



Date	Location	User	Time
12/16	Fastly HQ	Fred	2:30-5PM
12/18	Fastly HQ	Artur	10AM-12PM

Update function: insert 12/17/14, “Fastly HQ”, Ines

Dependency check: Is the requested time and location available?

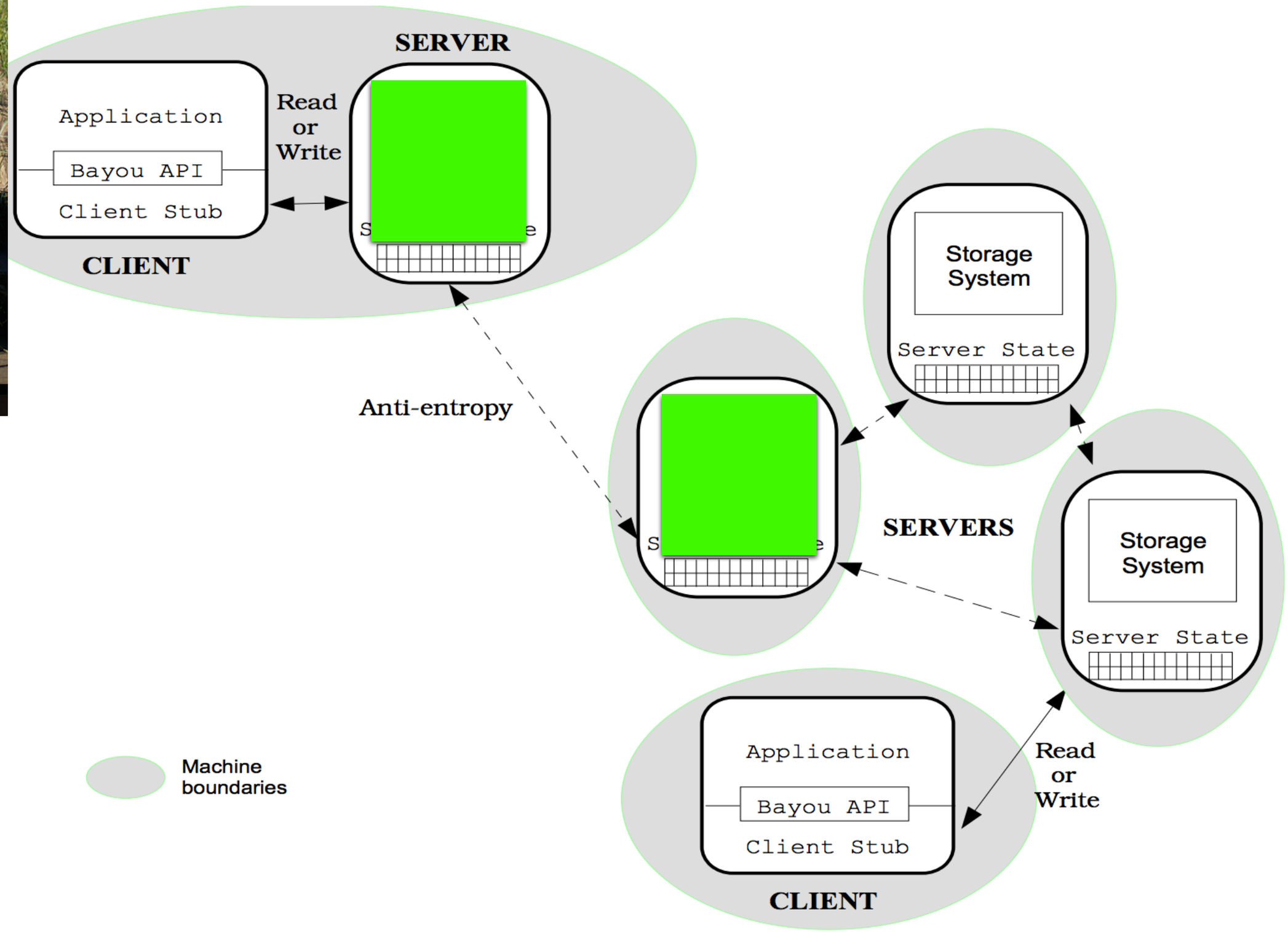
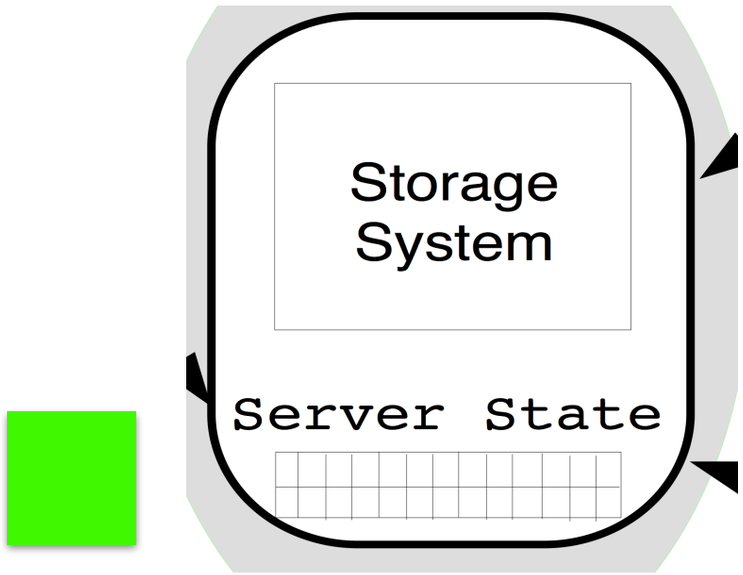
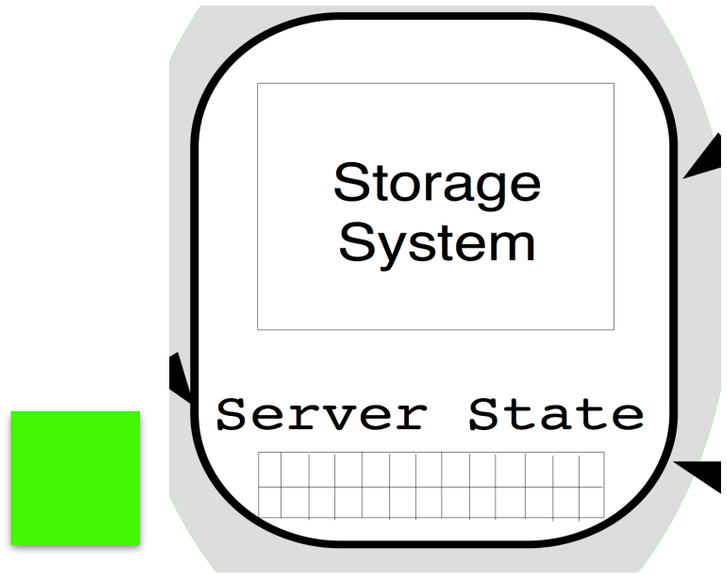
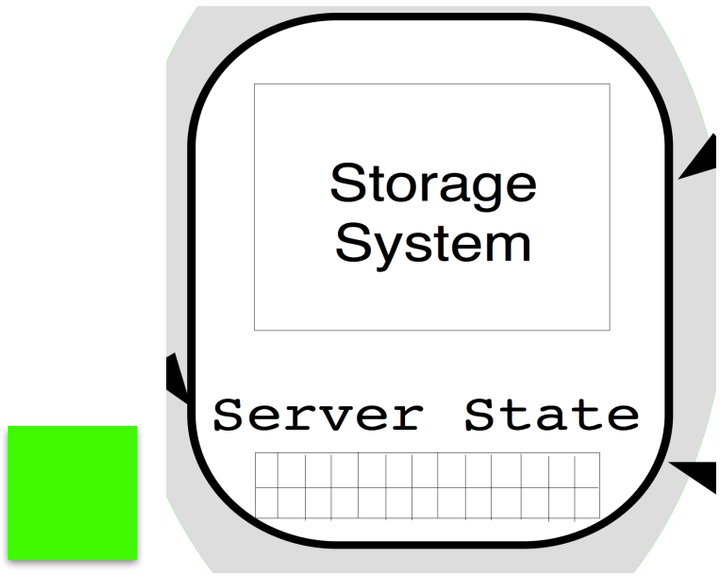
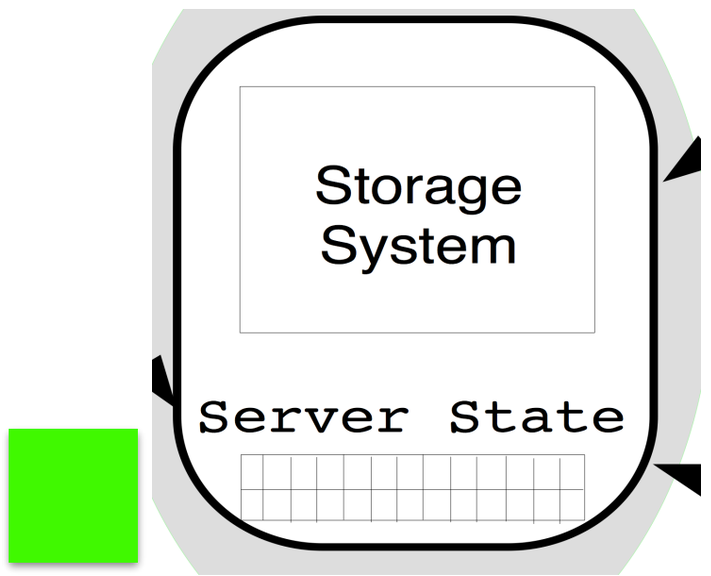


Figure 1. Bayou System Model


Storage
System
Server State


Storage
System
Server State


Storage
System
Server State


Storage
System
Server State

Another example:

Money transfer:

» Update function: transfer $100 from Jan to Mary

» Dependency check: does Jan have $100 in her account?

» Merge function: log an error

User	Balance
Jan	$110
Marsha	$10
Peter	$42

- Update function: transfer $100 from Jan to Marsha
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

Errors

User	Balance
Jan	$10
Marsha	$110
Peter	$42

Errors

- Update function: transfer $100 from Jan to Marsha
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

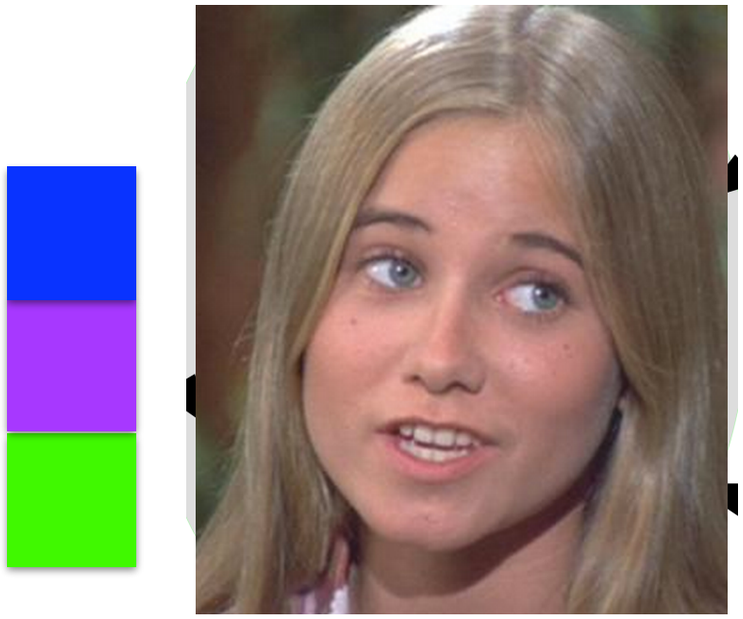
- Update function: transfer $100 from Jan to Peter
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

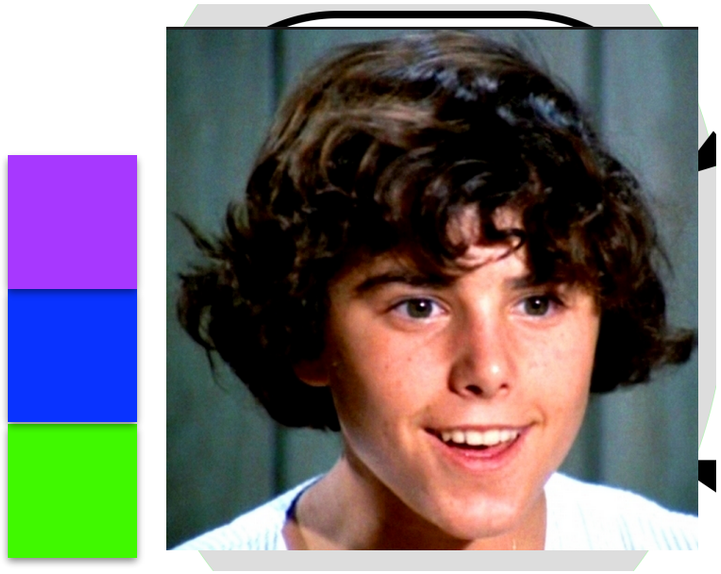
User	Balance
Jan	$10
Marsha	$110
Peter	$42

Errors
Jan->Peter failed!

- Update function: transfer $100 from Jan to Marsha
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

- Update function: transfer $100 from Jan to Peter
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error





Write stability

How do we ensure that all servers eventually agree?

Decide on a "stable" prefix of writes:

» Strawman: order writes by timestamp

- Drawbacks?

User	Balance
Jan	$10
Marsha	$110
Peter	$42

Errors

Timestamp 10

- Update function: transfer $100 from Jan to Marsha
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

Timestamp 2

- Update function: transfer $100 from Jan to Peter
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

User	Balance
Jan	$110
Marsha	$10
Peter	$42

Timestamp 2

- Update function: transfer $100 from Jan to Peter
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

Errors

User	Balance
Jan	$10
Marsha	$10
Peter	$142

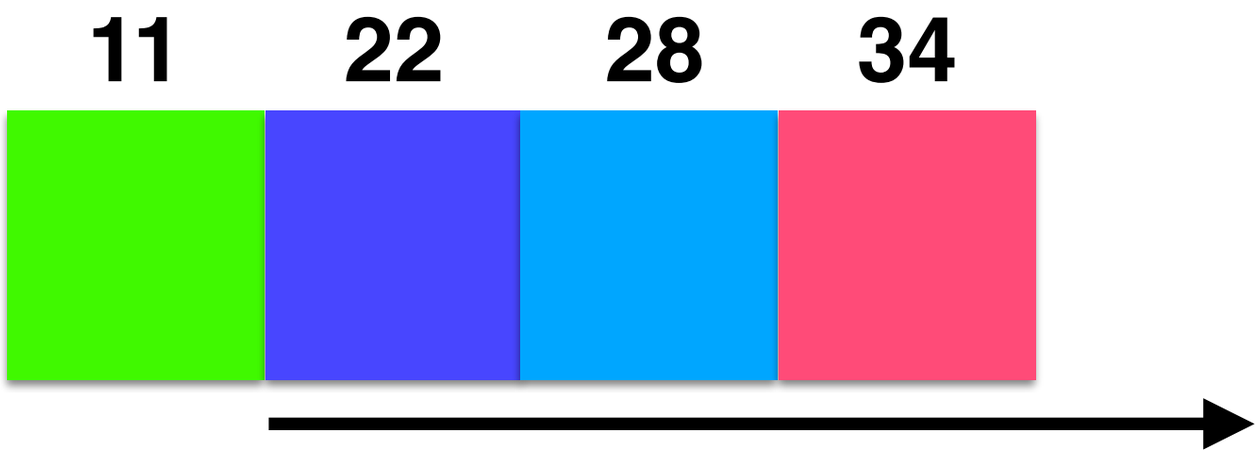
Errors
Jan->Marsha failed!

Timestamp 2

- Update function: transfer $100 from Jan to Peter
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

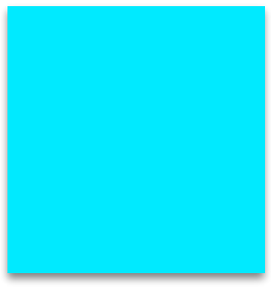
Timestamp 10

- Update function: transfer $100 from Jan to Marsha
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error



NEED TO RE-EXECUTE!

Also, makes GC hard



Write stability

How do we ensure that all servers eventually agree?

Decide on a "stable" prefix of writes:

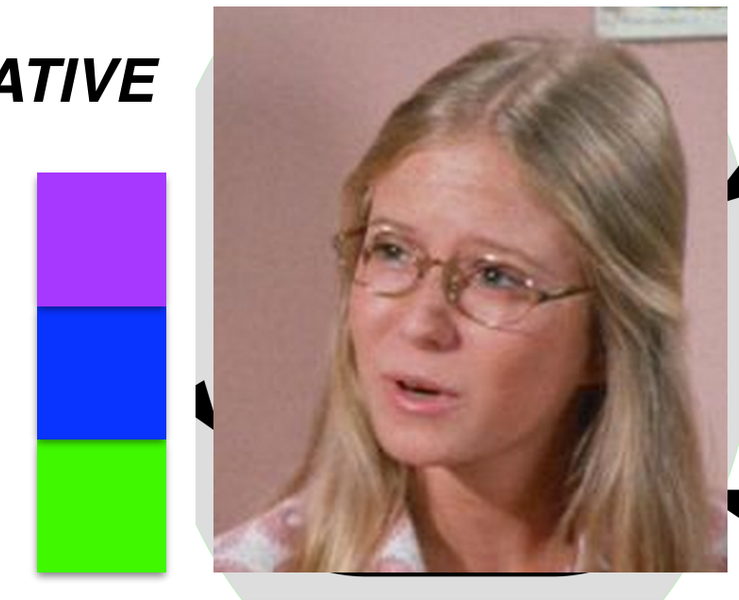
» Strawman: order writes by timestamp

» Bayou: uses master to determine ordering

T

C

TENTATIVE



COMMITTED

MASTER

T

C

T

C

Write stability

How do we ensure that all servers eventually agree?

Decide on a "stable" prefix of writes:

» Strawman: order writes by timestamp

- Drawbacks?

» Bayou: uses master to determine ordering

- Benefits?

什么意思?

Note: don't require commutative updates!

Read API

Can read from:

» Stable storage: only committed writes

» Tentative storage: all writes seen so far

Why read from tentative storage at all?

Session Guarantees for Weakly Consistent Replicated Data

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer,
and Brent B. Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304

Abstract

Four per-session guarantees are proposed to aid users and applications of weakly consistent replicated data: Read Your Writes, Monotonic Reads, Writes Follow Reads, and Monotonic Writes. The intent is to present individual applications with a view of the database that is consistent with their own actions, even if they read and write from various, potentially inconsistent servers. The guarantees can be layered on existing systems that employ a read-any/ write-any replication scheme while retaining the principal benefits of such a scheme, namely high-availability, simplicity, scalability, and support for disconnected operation. These session guarantees were developed in the context of the Bayou project at Xerox PARC in which we are designing and building a replicated storage system to support the needs of mobile computing users who may be only intermittently connected.

1. Introduction

Techniques for managing weakly consistent replicated data have been employed in a variety of systems [4,10,12,17,19,20]. Such systems are characterized by the lazy propagation of updates between servers and the possibility for clients to see inconsistent values when reading

may want to read and update data copied onto their portable computers even if they did not have the foresight to lock it before either a voluntary or an involuntary disconnection occurred. Also, the presence of slow or expensive communications links in the system can make maintaining closely synchronized copies of data difficult or uneconomical.

Unfortunately, the lack of guarantees concerning the ordering of read and write operations in weakly consistent systems can confuse users and applications, as reported in experiences with Grapevine [21]. A user may read some value for a data item and then later read an older value. Similarly, a user may update some data item based on reading some other data, while others read the updated item without seeing the data on which it is based. A serious problem with weakly consistent systems is that inconsistencies can appear even when only a single user or application is making data modifications. For example, a mobile client of a distributed database system could issue a write at one server, and later issue a read at a different server. The client would see inconsistent results unless the two servers had synchronized with one another sometime between the two operations.

In this paper, we introduce *session guarantees* that alleviate this problem of weakly consistent systems while maintaining the principle advantages of read-any/write-

DOI:10.1145/2500500

A broader class of consistency guarantees can, and perhaps should, be offered to clients that read shared data.

BY DOUG TERRY

Replicated Data Consistency Explained Through Baseball

REPLICATED STORAGE SYSTEMS for the cloud deliver different consistency guarantees to applications that are reading data. Invariably, cloud storage providers redundantly store data on multiple machines so that data remains available in the face of unavoidable

their applications.[5] These services ensure clients of Windows Azure Storage always see the latest value that was written for a data object. While strong consistency is desirable and reasonable to provide within a datacenter, it raises concerns as systems start to offer geo-replicated services that span multiple datacenters on multiple continents.

Many cloud storage systems, such as the Amazon Simple Storage Service (S3), were designed with weak consistency based on the belief that strong consistency is too expensive in large systems. The designers chose to relax consistency in order to obtain better performance and availability. In such systems, clients may perform read operations that return stale data. The data returned by a read operation is the value of the object at *some past point in time* but not necessarily the latest value. This occurs, for instance, when the read operation is directed to a replica that has not yet received all of the writes that were accepted by some other replica. Such systems are said to be *eventually consistent*.[12]

Recent systems, recognizing the need to support different classes of applications, have been designed with a choice of operations for accessing cloud storage. Amazon's DynamoDB, for example, provides both *eventually consistent reads* and *strongly consistent reads*, with the latter experiencing a higher read latency and a twofold reduction in read throughput.[1] Amazon SimpleDB offers the same choices for clients that

» key insights

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

Bayou's Secret Sauce

Push app logic into updates:

» Read and write are insufficiently expressive!

- Update function: transfer $100 from Jan to Peter
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

- Update function: transfer $100 from Jan to Mary
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

- **decrement Jan by $100; increment Peter by 100**
- **decrement Jan by $100; increment Mary by 100**

WRITE: Jan = 10; Peter = 42

WRITE: Jan = 10; Mary = 110

Bayou's Secret Sauce

Push app logic into updates:

» Read and write are insufficiently expressive!

Capture transaction intent:

» Dependency checks encode preconditions

» e.g., guarded atomic actions

What does Bayou guarantee?

Eventually all updates are applied in the same order on all servers

» Kind of like serializable/ACID transactions!

» Dependency checks enforce invariants

» Did we just "beat CAP"?

Key: **eventually** means we have to wait

H-Store, VoltDB, Granola, Calvin, Atomic Broadcast

Idea sketch: pre-schedule transactions, then execute them sequentially on respective servers

Totally ordered outcomes on each replica

…but ordering determined up front!

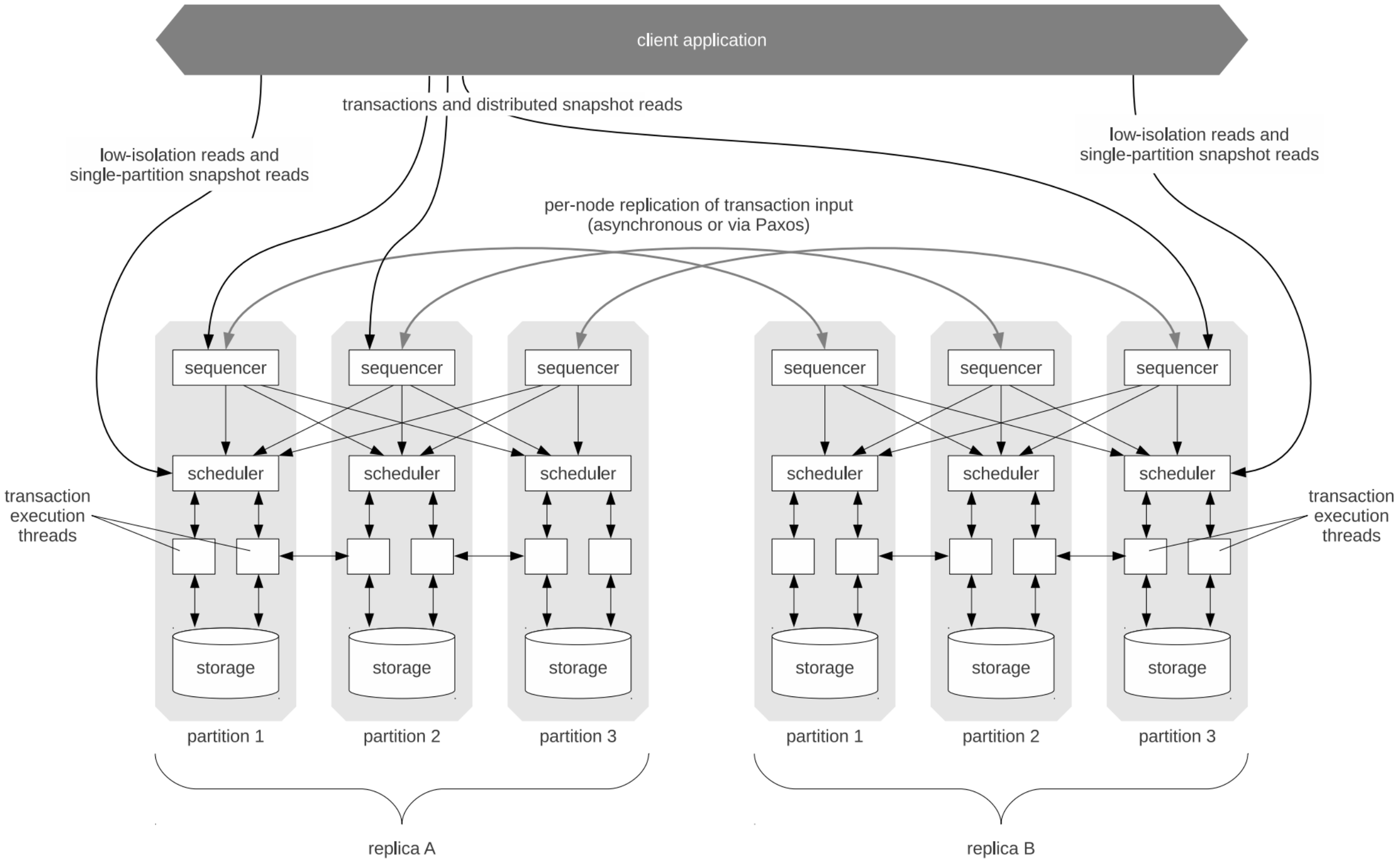


Figure 1: System Architecture of Calvin

Bayou ⇒ "ACID"

1.) Given transaction T, **issue write** W:

» Update function: // do nothing

» Dependency check: false

» Merge function: execute T

2.) Wait for write W **to commit.**

3.) Notify success.

Lazy Evaluation of Transactions in Database Systems

Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

Alexander Thomson
Google*
agt@google.com

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Existing database systems employ an *eager* transaction processing scheme—that is, upon receiving a transaction request, the system executes all the operations entailed in running the transaction (which typically includes reading database records, executing user-specified transaction logic, and logging updates and writes) before reporting to the client that the transaction has completed.

We introduce a *lazy* transaction execution engine, in which a transaction may be considered durably completed after only partial execution, while the bulk of its operations (notably all reads from the database and all execution of transaction logic) may be deferred until an arbitrary future time, such as when a user attempts to read some element of the transaction's write-set—all without modifying the semantics of the transaction or sacrificing ACID guarantees. Lazy transactions are processed deterministically, so that the final state of the database is guaranteed to be equivalent to what the state would have been had all transactions been executed eagerly.

Our prototype of a lazy transaction execution engine improves temporal locality when executing related transactions, reduces peak provisioning requirements by deferring more non-urgent work until off-peak load times, and reduces contention footprint of concurrent transactions. However, we find that certain queries suffer increased latency, and therefore lazy database systems may not be appropriate for read-latency sensitive applications.

the transaction, and then commits (or aborts). Upon receiving a query request, the database system performs the reads associated with the query and returns these results to the user or application that made the request. In both cases, the order is fixed: the database first performs the work associated with the transaction or query, and only afterwards does the database returns the results — read results, and/or the commit/abort decision.

In this paper, we explore the design of a database system that flips this traditional model on its head. For transactions that return only a commit/abort decision, the database first returns this decision and afterwards performs the work associated with that transaction. The meaning of "commit" and "abort" still maintain the full set of ACID guarantees: if the user is told that the transaction has committed, this means that the effects of the transaction are guaranteed to be durably reflected in the state of the database, and any subsequent reads of data that this transaction wrote will include the updates made by the committed transaction.

The key observation that makes this possible is inspired by the lazy evaluation research performed by the programming language community: the actual state of the database can be allowed to differ from the state of the database that has been promised to a client — it's only if the client actually makes an explicit request to read the state of the database do promises about state changes have to be kept. In particular, writes to the database state can be deferred, and "lazily" performed upon request — when a client reads the value

What does Bayou guarantee?

Eventually all writes are applied in the same order on all servers

» Kind of like serializable/ACID transactions!

» Dependency checks enforce invariants

» Did we just "beat CAP"?

Key: **eventually** means we have to wait 

"This meeting room scheduling program is intended for use after a group of people have already decided that they want to meet in a certain room and have determined a set of acceptable times for the meeting. It does not help them to determine a mutually agreeable place and time for the meeting, it only allows them to reserve the room."

"This meeting room scheduling program is intended for use **after a group of people have already decided** that they want to meet in a certain room and have determined a set of acceptable times for the meeting. It does not help them to determine a mutually agreeable place and time for the meeting, it only allows them to reserve the room."

"This meeting room scheduling program is intended for use after a group of people have already decided that they want to meet in a certain room and have determined a set of acceptable times for the meeting. It **does not help them to determine a mutually agreeable place and time for the meeting**, it only allows them to reserve the room."

When can we avoid “waiting”?

Commutative logic need not be re-executed in the log! (Paper discusses this.)

A note on commutativity

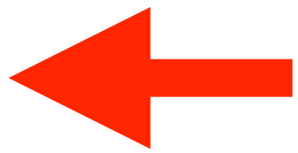
Commutative logic: decisions made in application logic are insensitive to ordering

Commutative datatypes: operations on datatypes are insensitive to ordering

- Update function: transfer $100 from Jan to Peter
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

- Update function: transfer $100 from Jan to Marsha
- Dependency check: does Jan have $100 in her account?
- Merge function: log an error

- **decrement Jan by $100; increment Peter by 100**
- **decrement Jan by $100; increment Marsha by 100**



WRITE: Jan = 10; Peter = 42

WRITE: Jan = 10; Marsha = 110

A note on commutativity

Commutative logic: decisions made in application logic are insensitive to ordering

Commutative datatypes: operations on datatypes are insensitive to ordering

The latter are useful, but won't ensure correctness!

http://www.bailis.org/blog/data-integrity-and-problems-of-scope/

When can we avoid “waiting”?

Commutative logic need not be re-executed in the log! (Paper discusses this.)

CALM Theorem: monotonic logic <=> determinism despite different orders [CIDR 2011] See also Kuper’s LVars

I-confluence: guarantees “safe” tentative reads with convergent and safe outcomes [VLDB 2015]

A note on immutability

Bayou's stable log is "immutable"

» e.g., event sourcing, Lambda architecture

But what guarantees can we make about the outcomes in the log?

» "Immutable" writes are the easy part!

» Reasoning about outcomes is the challenge

Why have "merge" at all?

Why not just ship the stored procedures and re-execute them instead?

» Update function: transfer $100 from Jan to Peter

» Dependency check: does Jan have $100 in her account?

If Jan has $100:
    transfer $100 from Jan to Peter
Else:
    Log an error

Eventually-Serializable Data Services

Alan Fekete* David Gupta† Victor Luchangco† Nancy Lynch† Alex Shvartsman†

Abstract

We present a new specification for distributed data services that trade-off immediate consistency guarantees for improved system availability and efficiency, while ensuring the long-term consistency of the data. An *eventually-serializable* data service maintains the operations requested in a partial order that gravitates over time towards a total order. It provides clear and unambiguous guarantees about the immediate and long-term behavior of the system. To demonstrate its utility, we present an algorithm, based on one of Ladin, Liskov, Shrira, and Ghemawat [12], that implements this specification. Our algorithm provides the interface of the abstract service, and generalizes their algorithm by allowing general operations and greater flexibility in specifying consistency requirements. We also describe how to use this specification as a building block for applications such as directory services.

1 Introduction

Providing distributed and concurrent access to data objects is a fundamental concern of distributed systems. The simplest implementations maintain a single centralized object that is accessed remotely by multiple clients. While conceptually simple, this approach does not scale well as the number of clients increases. To address this

Replication of the data object raises the issue of consistency among the replicas, especially in determining the order in which the operations are applied at each replica. The strongest and simplest notion of consistency is *atomicity*, which requires the replicas to collectively emulate a single centralized object. Methods to achieve atomicity include write-all/read-one [4], primary copy [1, 21, 18], majority consensus [22], and quorum consensus [8, 9]. Because achieving atomicity often has a high performance cost, some applications, such as directory services, are willing to tolerate some transient inconsistencies. This gives rise to different notions of consistency. *Sequential consistency* [13], guaranteed by systems such as Orca [3], allows operations to be reordered as long as they remain consistent with the view of individual clients. An inherent disparity in the performance of atomic and sequentially consistent objects has been established [2]. Other systems provide even weaker guarantees to the clients [6, 5, 7] in order to get better performance.

Improving performance by providing weaker guarantees results in more complicated semantics. Even when the behavior of the replicated objects is specified unambiguously, it is more difficult to understand and to reason about the correctness of implementations. In practice, replicated systems are often incompletely or ambiguously specified.

Edelweiss: Automatic Storage Reclamation for Distributed Programming

Neil Conway, Peter Alvaro, Emily Andrews, Joseph M. Hellerstein
UC Berkeley
{nrc, palvaro, e.andrews, hellerstein}@cs.berkeley.edu

ABSTRACT

Event Log Exchange (ELE) is a common programming pattern based on immutable state and messaging. ELE sidesteps traditional challenges in distributed consistency, at the expense of introducing new challenges in designing space reclamation protocols to avoid consuming unbounded storage.

We introduce *Edelweiss*, a sublanguage of Bloom that provides an ELE programming model, yet automatically reclaims space without programmer assistance. We describe techniques to analyze Edelweiss programs and automatically generate application-specific distributed space reclamation logic. We show how Edelweiss can be used to elegantly implement a variety of communication and distributed storage protocols; the storage reclamation code generated by Edelweiss effectively garbage-collects state and often matches hand-written protocols from the literature.

1. INTRODUCTION

> *"Blossom of snow may you bloom and grow,*
> *bloom and grow forever."*
> —Oscar Hammerstein, "Edelweiss"

Distributed and parallel systems are increasingly commonplace, but writing reliable programs for these environments remains stubbornly difficult. Both developers and academics have identified *shared mutable state* as a common source of problems: code that mutates shared state is hard to reason about and often requires expensive, fine-grained synchronization and careful coordination between

By using event logs, ELE designs achieve a variety of familiar benefits from database research. For example, rather than determining a conservative order for modifications to shared state, ELE can allow operations to be applied in different orders at different replicas and reconciled later, reducing the need for coordination and increasing concurrency and availability. ELE allows simple mechanisms for fault tolerance and recovery via log replay, and provides a natural basis for system debugging and failure analysis.

ELE is an attractive approach to simplifying distributed programming, but it introduces its own complexities. If each process accumulates knowledge over time, the required storage will grow without bound. To avoid this, ELE designs typically include a background "garbage collection" or "checkpointing" protocol that reclaims information that is no longer useful. This pattern of logging and background reclamation is widespread in the distributed storage and data management literature, having been applied to many core techniques including reliable broadcast and update propagation [13, 14, 19, 22, 28, 34, 38, 45], group communication [17], key-value storage [3, 15, 45], distributed file systems [18, 26, 35], causal consistency [7, 23, 27], quorum consensus [21], transaction management [2, 11], and multi-version concurrency control [36, 39, 44].

Despite the similarity of these example systems, each design typically includes a storage reclamation scheme that has been developed from scratch and implemented by hand. Such schemes can be subtle and hard to get right: reclaiming garbage too eagerly is unsafe (because live data is incorrectly discarded), whereas an overly conservative scheme can result in hard-to-find resource leaks. Moreover, the conditions under which stored values can be reclaimed depends on program semantics; hence, a hand-crafted garbage collection procedure must be updated as the program is evolved, making software

Durability?

In this context, durability and consistency are effectively orthogonal

- » Durability: "survive F faults, need F+1 servers"
- » Strong consistency: usually requires "majority"

Bayou: local updates may not survive faults

- » But the (arguably) more fundamental part of the system is maintaining updates

Bayou vs. Dynamo

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

Talk Outline

Background and system architecture

Conflict detection and resolution APIs

“Correctness” and ordering

Lessons for all of us

Why Bayou?

"NoSQL" before "NoSQL"

Learn from smart predecessors

Rethink application, system boundaries

Lucid discussion of systems challenges

Bayou Goals

Handle frequent disconnections

» Embrace replication, update-anywhere

Reason about distribution explicitly

» Require application semantics

Facilitate conflict detection and resolution

» Use merge and dependency APIs

"Weakly consistent replication has been used previously for availability, simplicity, and scalability in a variety of systems [3, 7, 10, 12, 15, 19]."

"Weakly consistent replication has been used previously for availability, **simplicity**, and scalability in a variety of systems [3, 7, 10, 12, 15, 19]."

Simplicity for whom?

Architects

Systems programmers

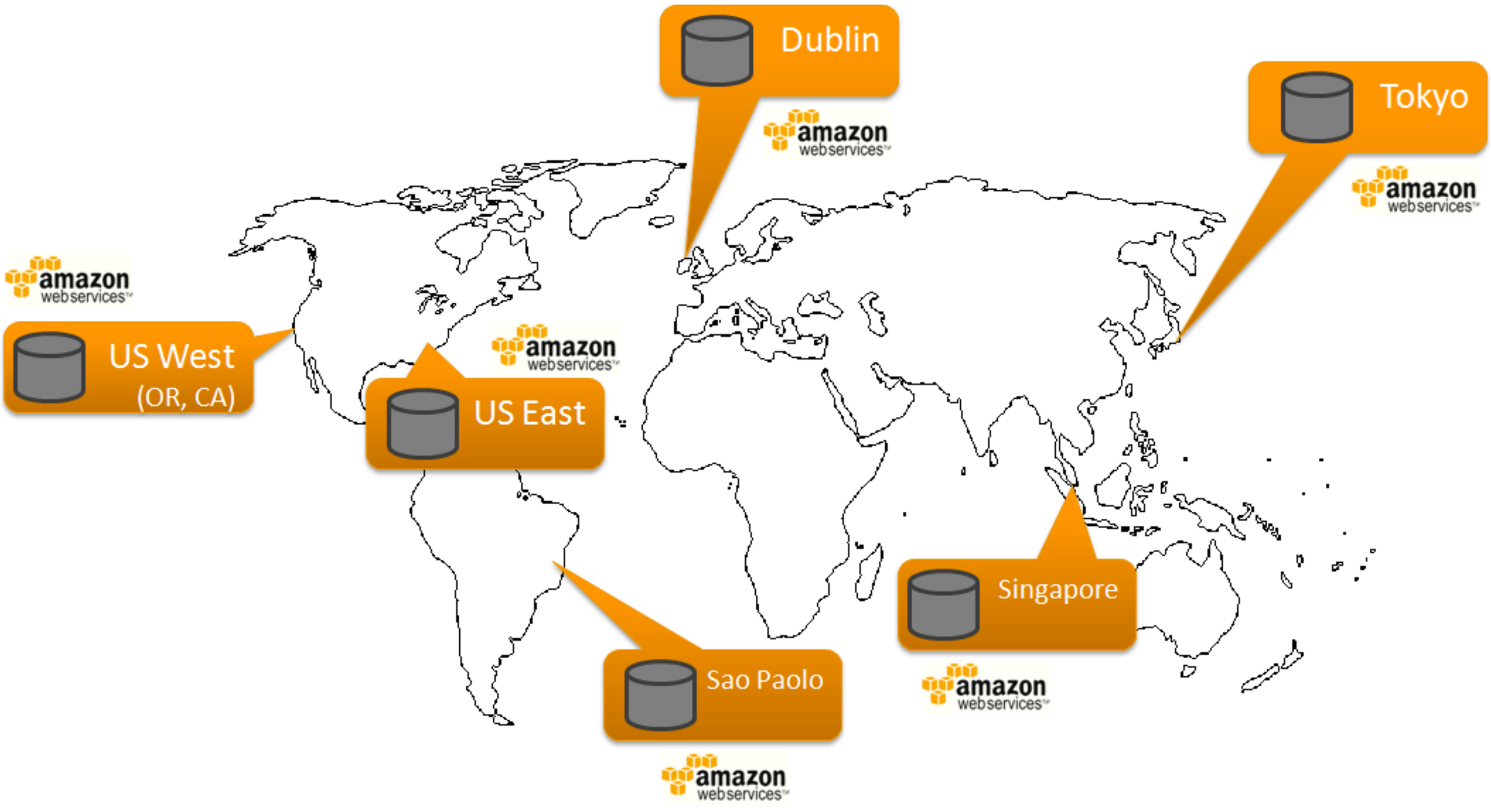
Operators

Application writers?

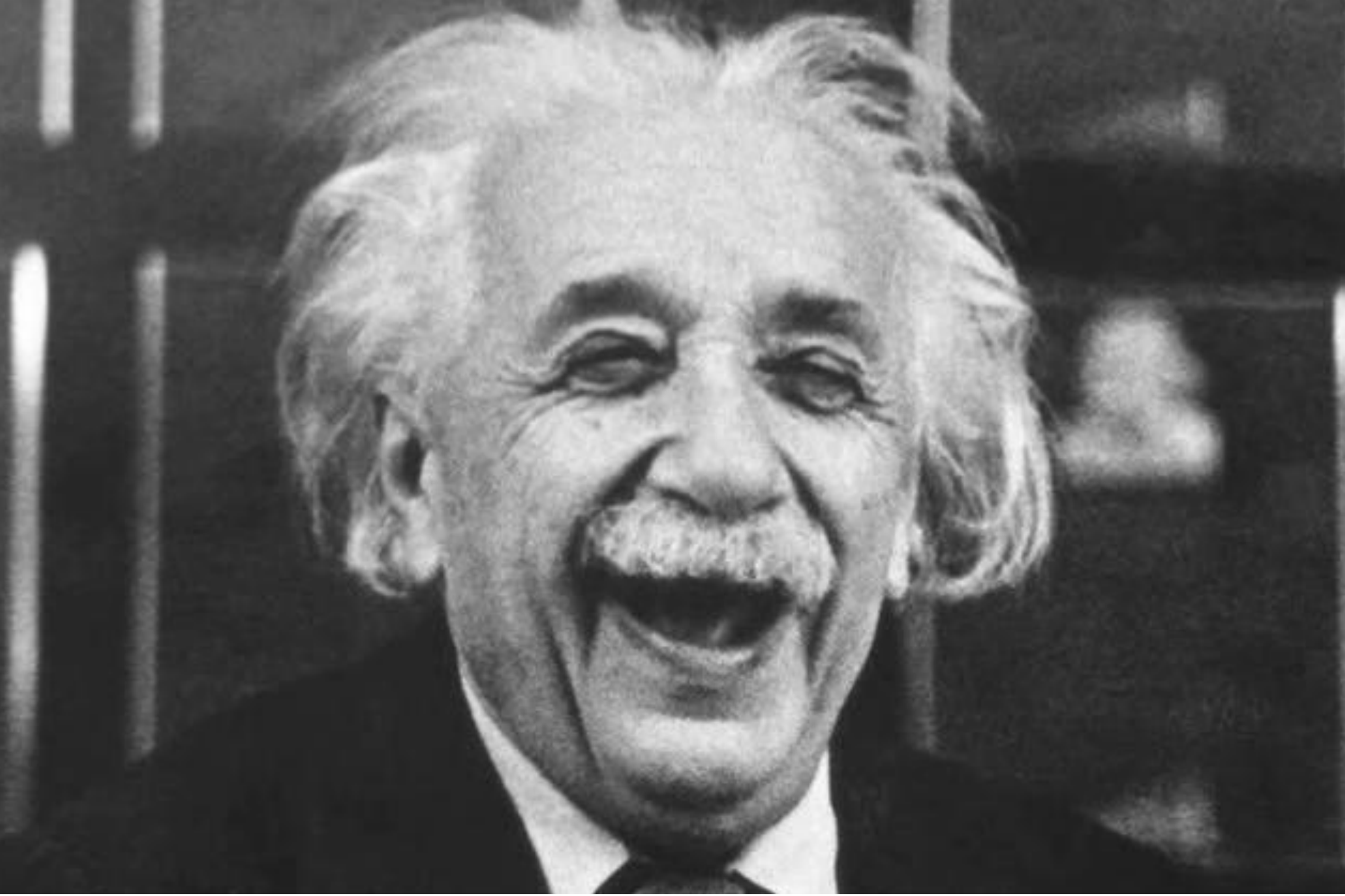
Users?

Analogous issues in RDBMS design!

http://www.bailis.org/blog/understanding-weak-isolation-is-a-serious-problem/



Unfortunately, not always a choice!



THOSE LIGHT CONES

2.6 Billion Internet users in 2013

7.1 Billion humans on planet Earth [Mary Meeker]

Who cares about scale? Coordination?

Will data keep increasing? Why?

Hint: not humans.

AT&T M-Cell

Home

Tweet

In reply to (null)

Cliff Moon

@moonpolysoft

@strlen really? I thought southbay was all about office parks and strip malls.

3 hours ago via web

Replies


NEVADA
001
AUTONOMOUS VEHICLE
TOYOTA
PRIUS
Google™
self-driving car

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 roll-back the effects of previously executed writes and redo them according to a global serialization order. Furthermore, Bayou permits clients to observe the results of all writes received by a server, including tentative writes whose conflicts have not been ultimately resolved. This paper presents the motivation for and design of these mechanisms and describes the experiences gained with an initial implementation of the system.

1. Introduction

The Bayou storage system provides an infrastructure for collaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. The advent of mobile computers, in the form of laptops and personal digital assistants (PDAs) enables the use of computational facilities away from the usual "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 clients can read and write to any replica without the need for explicit coordination with other replicas. Every computer eventually receives updates from every other, either directly or indirectly, through a chain of pair-wise interactions.

Unlike many previous systems [12, 27], our goal in designing the Bayou system was *not* to provide transparent replicated data support for existing file system and database applications. We believe that applications must be aware that they may read weakly consistent data and also that their write operations may conflict with those of other users and applications. Moreover, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application.

To this end, Bayou provides system support for application-specific conflict detection and resolution. Previous systems, such as Locus [30] and Coda [17], have proven the value of semantic

Session Guarantees for Weakly Consistent Replicated Data

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer,
and Brent B. Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304

Abstract

Four per-session guarantees are proposed to aid users and applications of weakly consistent replicated data: Read Your Writes, Monotonic Reads, Writes Follow Reads, and Monotonic Writes. The intent is to present individual applications with a view of the database that is consistent with their own actions, even if they read and write from various, potentially inconsistent servers. The guarantees can be layered on existing systems that employ a read-any/ write-any replication scheme while retaining the principal benefits of such a scheme, namely high-availability, simplicity, scalability, and support for disconnected operation. These session guarantees were developed in the context of the Bayou project at Xerox PARC in which we are designing and building a replicated storage system to support the needs of mobile computing users who may be only intermittently connected.

1. Introduction

Techniques for managing weakly consistent replicated data have been employed in a variety of systems [4,10,12,17,19,20]. Such systems are characterized by the lazy propagation of updates between servers and the possibility for clients to see inconsistent values when reading

may want to read and update data copied onto their portable computers even if they did not have the foresight to lock it before either a voluntary or an involuntary disconnection occurred. Also, the presence of slow or expensive communications links in the system can make maintaining closely synchronized copies of data difficult or uneconomical.

Unfortunately, the lack of guarantees concerning the ordering of read and write operations in weakly consistent systems can confuse users and applications, as reported in experiences with Grapevine [21]. A user may read some value for a data item and then later read an older value. Similarly, a user may update some data item based on reading some other data, while others read the updated item without seeing the data on which it is based. A serious problem with weakly consistent systems is that inconsistencies can appear even when only a single user or application is making data modifications. For example, a mobile client of a distributed database system could issue a write at one server, and later issue a read at a different server. The client would see inconsistent results unless the two servers had synchronized with one another sometime between the two operations.

In this paper, we introduce *session guarantees* that alleviate this problem of weakly consistent systems while maintaining the principle advantages of read-any/write-

Flexible Update Propagation for Weakly Consistent Replication

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

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

Abstract

Bayou's anti-entropy protocol for update propagation between weakly consistent storage replicas is based on pair-wise communication, the propagation of write operations, and a set of ordering and closure constraints on the propagation of the writes. The simplicity of the design makes the protocol very flexible, thereby providing support for diverse networking environments and usage scenarios. It accommodates a variety of policies for when and where to propagate updates. It operates over diverse network topologies, including low-bandwidth links. It is incremental. It enables replica convergence, and updates can be propagated using floppy disks and similar transportable media. Moreover, the protocol handles replica creation and retirement in a light-weight manner. Each of these features is enabled by only one or two of the protocol's design choices, and can be independently incorporated in other systems. This paper presents the anti-entropy protocol in detail, describing the design decisions and resulting features.

1. Introduction

Weakly consistent replicated storage systems with an "update anywhere" model for data modifications require a protocol for replicas to reconcile their state, that is, a protocol to propagate the updates introduced at one replica to all other replicas. A key advantage of weakly consistent replication is that, by relaxing data consistency, the protocol for data propagation can accommodate policy choices for *when* to reconcile, *with whom* to reconcile, and even *what* data to reconcile. In this paper we present Bayou's **anti-entropy** protocol for replica reconciliation. The protocol, while simple in design, has several features intended to support diverse network environments and usage scenarios. The contribution of this paper is to demonstrate how

- *Incremental progress*:
  the protocol allows incremental progress even if interrupted, for example, due to an involuntary network disconnection.
- *Eventual consistency*:
  each update eventually reaches every replica, and replicas holding the same updates have the same database contents.
- *Efficient storage management*:
  the protocol allows replicas to discard logged updates to reclaim storage resources used for reconciliation.
- *Propagation through transportable media:*
  one replica can send updates to another by storing the updates on transportable media, like diskettes, without ever having to establish a physical network connection.
- *Light-weight management of dynamic replica sets*:
  the protocol supports the creation and retirement of a replica through communication with only one available replica.
- *Arbitrary policy choices*:
  any policy choices for when to reconcile and with which replicas to reconcile are supported by the anti-entropy mechanism. The policy need only ensure that there be an eventual communication path between any pair of replicas.

Other weakly consistent replicated systems support subsets of these functionalities. For example, Coda's reconciliation protocols allow server replicas to reconcile with each other, and mobile replicas to reconcile with servers, but mobiles cannot reconcile amongst themselves [11]. In Ficus, reconciliation can occur between any pair of replicas, however server creation and retirement requires coordination among all replicas [7]. Oracle 7 has a two-level hierarchy of replicas: master replicas send their transactions to all other masters, but cannot forward transactions

The Bayou Architecture: Support for Data Sharing among Mobile Users

Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, Brent Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.
contact: terry@parc.xerox.com

Abstract

The Bayou System is a platform of replicated, highly-available, variable-consistency, mobile databases on which to build collaborative applications. This paper presents the preliminary system architecture along with the design goals that influenced it. We take a fresh, bottom-up and critical look at the requirements of mobile computing applications and carefully pull together both new and existing techniques into an overall architecture that meets these requirements. Our emphasis is on supporting application-specific conflict detection and resolution and on providing application-controlled inconsistency.

1. Introduction

The Bayou project at Xerox PARC has been designing a system to support data sharing among mobile users. The system is intended to run in a mobile computing environment that includes portable machines with less than ideal network connectivity. In particular, a user's computer may have a wireless communication device, such as a cell modem or packet radio transceiver relying on a network infrastructure that is not universally available and perhaps unreasonably expensive. It may use short-range line-of-

We believe that mobile users want to share their appointment calendars, bibliographic databases, meeting notes, evolving design documents, news bulletin boards, and other types of data in spite of their intermittent network connectivity. The focus of the Bayou project has been on exploring mechanisms that let mobile clients actively read and write shared data. Even though the system must cope with both voluntary and involuntary communication outages, it should look to users, to the extent possible, like a centralized, highly-available database service. This paper presents detailed goals for the overall system architecture and discusses the design decisions that we made to meet these goals.

2. Architectural design decisions

Goal: Support for portable computers with limited resources.

Design: A flexible client-server architecture.

Many of the devices that we envision being commonly used, such as PDAs and the ParcTab developed within our lab [24], have insufficient storage for holding copies of all, or perhaps any, of the data that their users want to access. For this reason, our architecture is based

Xerox PARC

Laser printers

Computer-generated bitmap graphics

The Graphical user interface, featuring windows and icons, operated with a mouse

The WYSIWYG text editor

The precursor to PostScript

Ethernet as a local-area computer network

Fully formed object-oriented programming in the Smalltalk programming language and integrated development environment.

Model–view–controller software architecture

Bayou!

[Wikipedia]

Xerox PARC

DEC SRC

MSR SVC

???

What can we learn?

1.) Integrating application logic is key to "correct" execution of "AP" distributed systems. R/W is bad.

2.) Lack of app-specific mechanisms in a coordination-free system is a recipe for data corruption.

3.) Merge/repair is a narrow API for developers to express their application conflicts.

4.) Alternatives like commutativity, I-confluence, and research tools like Bloom can help limit overhead.

Thanks Fastly & Ines!