Joining Forces:
Toward a Unified Account of LVars and Convergent Replicated Data Types

Lindsey Kuper and Ryan Newton
Indiana University

WoDet ’14, Salt Lake City, UT
March 2, 2014
Parallel systems

Distributed systems
Parallel systems

$LVars$

Distributed systems
Parallel systems

$LVars$

Distributed systems

$CvRDTs$
(Convergent Replicated Data Types)
Parallel systems

Distributed systems

$LVars \join CvRDTs$

(Convergent Replicated Data Types)
| Item | Book | Shoes | ... |
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
\textbf{data} Item = Book \mid Shoes \mid \ldots
\begin{verbatim}
p :: IO (Map Item Int)
p = do cart <- newIORef empty
\end{verbatim}
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
     async (atomicModifyIORef cart
             (\m -> (insert Book 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
           (\m -> (insert Book 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
      async (atomicModifyIORef cart
             (\m -> (insert Book 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)

p = do cart <- newIORef empty
    async (atomicModifyIORef cart
          (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
          (\m -> (insert Shoes 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
     async (atomicModifyIORef cart
     (\m -> (insert Book 1 m, ())))
     async (atomicModifyIORef cart
     (\m -> (insert Shoes 1 m, ()))))
     res <- async (readIORef cart)
     wait res
landin:ivar-examples lkuper$ make map-iorref-data-race
ghc -O2 map-iorref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main
   ( map-iorref-data-race.hs, map-iorref-data-race.o )
Linking map-iorref-data-race ...
while true; do ./map-iorref-data-race +RTS -N2; done
landin:ivar-examples lkuper$ make map-ioref-data-race
ghc -02 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main
  ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true: do /map-ioref-data-race +RTS -N2; done

((Book,1),(Shoes,1))
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart (\m -> (insert Book 1 m, ()))))
  a2 <- async (atomicModifyIORef cart (\m -> (insert Shoes 1 m, ()))))
  res <- async (do waitBoth a1 a2
                   readIORef cart)
    wait res
main = do v <- p
          print v
Deterministic...now
Deterministic...now...we hope
Deterministic...now...we hope
data Item = Book | Shoes | ...

p = do
   cart <- newEmptyMap
   fork (insert Shoes 1 cart)
   fork (insert Book 2 cart)
   getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ... 

p = do 
cart <- newEmptyMap 
fork (insert Shoes 1 cart) 
fork (insert Book 2 cart) 
getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
    cart <- newEmptyMap
    fork (insert Shoes 1 cart)
    fork (insert Book 2 cart)
    getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
    cart <- newEmptyMap
    fork (insert Shoes 1 cart)
    fork (insert Book 2 cart)
    getKey Book cart -- returns 2
\[
\text{data Item} = \text{Book} \mid \text{Shoes} \mid \ldots \\
\{(\text{Book},1), (\text{Book},2), \ldots\}
\]

\[
p = \text{do} \\
cart \leftarrow \text{newEmptyMap} \\
fork (\text{insert Shoes 1 cart}) \\
fork (\text{insert Book 2 cart}) \\
\text{getKey Book cart} -- \text{returns 2}
\]
\textbf{data} Item = Book | Shoes | ... \\

\( p = \text{do} \)
\begin{align*}
& \text{cart} \leftarrow \text{newEmptyMap} \\
& \text{fork} (\text{insert Shoes 1 cart}) \\
& \text{fork} (\text{insert Book 2 cart}) \\
& \text{getKey Book cart} \quad \text{-- returns 2}
\end{align*}
The threshold set must be \textit{pairwise incompatible}.

\begin{verbatim}
\{ (Book, 1), (Book, 2), \ldots \}
\end{verbatim}

\begin{itemize}
\item data Item = Book | Shoes | \ldots
\item p = do
    cart <- newEmptyMap
    fork (insert Shoes 1 cart)
    fork (insert Book 2 cart)
    getKey Book cart -- returns 2
\end{itemize}
The threshold set must be pairwise incompatible.

```haskell
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
```

hackage.haskell.org/package/lvish
Parallel systems

$LVars$

Distributed systems

$CvRDTs$

(Convergent Replicated Data Types)
Eventual consistency.
Eventual consistency. How?
Dynamo: Amazon’s Highly Available Key-value Store

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

Amazon.com

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

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

Categories and Subject Descriptors
D.4.2 [Operating Systems]: Storage Management; D.4.5 [Operating Systems]: Reliability; D.4.2 [Operating Systems]: Performance;

General Terms

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

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

SIGOPS’07 October 14-17, 2007, Stevensen, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

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

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

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

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

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs
Dynamo: Amazon’s Highly Available Key-value Store
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels
Amazon.com

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

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

Categories and Subject Descriptors

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

DeCandia et al., SOSP ’07
Conflict-Free Replicated Data Types*

Marc Shapiro1,2, Nuno Preguiça1,2, Carlos Baquero3, and Marek Zawirski1,4
1 INRIA, Paris, France
2 CITI, Universidade Nova de Lisboa, Portugal
3 Universidade do Minho, Portugal
4 UPMC, Paris, France

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

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

1 Introduction

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

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, eventual consistency promises better availability and performance [12][11]. An update executes at some replica, without synchronisation; later, it is sent to the other replicas or removed at the publisher. No updates are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy, republish, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

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

To meet the requirements for scaling needs, Amazon has developed Dynamo, another highly available distributed key-value store inside of Amazon and known as Dynamo. This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon’s core services use to provide an “always-on” experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-supported fault resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

Dynamo: Amazon’s Highly Available Key-value Store

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

Amazon.com

Dynamo is aware of the data schema it manages, and has a method that is best suited for each type. The application that maintains the shopping cart needs to be able to “merge” the conflicting updates into a single shopping cart.

In this paper, we present the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon’s core services use to provide a “always-on” experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-supported fault resolution in a manner that provides a novel interface for developers to use.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability. Data is partitioned and replicated using consistent hashing [9], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs
Dynamo: Amazon’s Highly Available Key-value Store

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

Amazon.com

ABSTRACT

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a firm is dependent on how its application state is managed. For example, applications use a highly decentralized, loosely coupled service architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to save and retrieve items in their shopping cart even if they are not in the same data center. The service responsible for storing shopping cart requires that it can always write to and read from its data store, and that its data needs to be available at multiple data centers.

The key insight is that the data at each replica is aware of the data schema it follows. It is possible to build a method that is best suited for this, the application that maintains the data. The technique under consideration for “merge” the conflicting updates to the replicated shopping cart.

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

[Shapiro et al., SOSP ’07]
Two “styles” of Conflict-Free Replicated Data Types:

“Convergent”
CvRDTs
“state-based”

“Commutative”
CmRDTs
“op-based”

\[
\text{put} \cdot \text{put} = \text{put} \cdot \text{put}
\]
Two “styles” of Conflict-Free Replicated Data Types:

“Convergent”
CvRDTs
“state-based”

“Commutative”
CmRDTs
“op-based”

[Shapiro et al., SSS '11]
Two “styles” of Conflict-Free Replicated Data Types:

“Convergent” CvRDTs
“state-based”

“Commutative” CmRDTs
“op-based”

[Shapiro et al., SSS ’11]

\[\text{put} \cdot \text{put} \Rightarrow \text{put} \cdot \text{put} = \text{put} \cdot \text{put}\]
## LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>LVars</th>
<th>CvRDTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold reads (deterministic)</td>
<td>Ordinary reads (nondeterministic)</td>
</tr>
<tr>
<td>Least-upper-bound writes (every write computes a join)</td>
<td>General inflationary writes (only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>
LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads (deterministic)</th>
<th>Ordinary reads (nondeterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-upper-bound writes (every write computes a join)</td>
<td>General inflationary writes (only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>
LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads</th>
<th>Ordinary reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>(deterministic)</td>
<td>(nondeterministic)</td>
</tr>
<tr>
<td>Least-upper-bound writes</td>
<td>General inflationary writes</td>
</tr>
<tr>
<td>(every write computes a join)</td>
<td>(only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>
LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads (deterministic)</th>
<th>Ordinary reads (nondeterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-upper-bound writes</td>
<td>General inflationary writes</td>
</tr>
<tr>
<td>(every write computes a join)</td>
<td>(only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>

so we're proposing
Adding threshold reads to CvRDTs

- **Threshold reads (deterministic)**
- **Ordinary reads (nondeterministic)**

- Least-upper-bound writes (every write computes a join)
- General inflationary writes (only merges must be joins)

- Shared memory
- Replicated!

Adding threshold reads to CvRDTs
One framework for reasoning about both eventual and strong consistency

so we’re proposing
### LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads (deterministic)</th>
<th>Ordinary reads (nondeterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-upper-bound writes (every write computes a join)</td>
<td>General inflationary writes (only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>

- Adding threshold reads to CvRDTs
  One framework for reasoning about both eventual and strong consistency
- Adding general inflationary writes to LVars
  Non-idempotent, incrementable counters

so we’re proposing
Thank you!

Email: lkuper@cs.indiana.edu
LVars project repo: github.com/iu-parfunc/lvars
LVars papers: cs.indiana.edu/~lkuper
Research blog: composition.al

- Adding threshold reads to CvRDTs
  One framework for reasoning about both eventual and strong consistency
- Adding general inflationary writes to LVars
  Non-idempotent, incrementable counters

so we’re proposing