Consistency choices in modern distributed systems

Alexey Gotsman

IMDEA Software Institute, Madrid, Spain
Data is replicated and partitioned across multiple nodes
Data centres across the world

Disaster-tolerance, minimising latency
Data centres across the world

Disaster-tolerance, minimising latency
Data centres across the world

Disaster-tolerance, minimising latency
With thousands of machines inside

Load-balancing, fault-tolerance
Replicas on mobile devices

Offline use
- **Strong consistency model**: the system behaves as if it processes requests serially on a centralised database - linearizability, serializability
• **Strong consistency model:** the system behaves as if it processes requests serially on a centralised database - linearizability, serializability

• Requires **synchronisation:** contact other replicas when processing a request
• Expensive: communication increases latency

• Impossible: either strong Consistency or Availability in the presence of network Partitions [CAP theorem]
- Expensive: communication increases latency
- Impossible: either strong Consistency or Availability in the presence of network Partitions [CAP theorem]
Relaxing synchronisation

Process an update locally, propagate effects to other replicas later
Relaxing synchronisation

Process an update locally, propagate effects to other replicas later

- Better scalability & availability

- Weakens consistency: deposit seen with a delay
Anomalies

add(100)

notify(done)
Anomalies

add(100)

notify(done)

getNotif() : done

balance() : 0
Anomalies

Causal dependency: one operation is aware of another
Causal consistency model: disallows this anomaly
Weak consistency elsewhere

- Centralised DBs: snapshot isolation, read committed, ...
Weak consistency elsewhere

- Centralised DBs: snapshot isolation, read committed, ...
- Multicore processors: x86, ARM
Weak consistency elsewhere

- Centralised DBs: snapshot isolation, read committed, ...
- Multicore processors: x86, ARM
- Programming languages: C/C++, Java
Programming models

• Replicated data types (aka CRDTs): encapsulate conflict-resolution policies

• Novel forms of transactions

• Varying consistency among operations
Programming models

• Replicated data types (aka CRDTs): encapsulate conflict-resolution policies

• Novel forms of transactions

• Varying consistency among operations
Problem

Poor guidelines on how to use the new programming models: are we weakening consistency too much, too little, just right?
Problem

Poor guidelines on how to use the new programming models: are we weakening consistency too much, too little, just right?

"If no new updates are made to the database, then replicas will eventually reach a consistent state"
Problem

Poor guidelines on how to use the new programming models: are we weakening consistency too much, too little, just right?

“If no new updates are made to the database, then replicas will eventually reach a consistent state”

Eventually Consistent

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

By Werner Vogels

At the foundation of Amazon’s cloud computing are infrastructure services such as Amazon’s S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost-effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customer of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when many widespread distributed systems provide an eventual consistency model in the context of data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. Here, I present some of the intricate background that has informed our approach to delivering reliable distributed systems that must operate on a global scale. (An earlier version of this article appeared as a posting on the “All Things Distributed” Weblog and was greatly improved with the help ofCommentators.)

Historical Perspective

In an ideal world there would be only one consistency model: when an update is made all observers would see that update. The first time this surfaced as an issue that needed to be addressed was in the database systems of the late 1970s. The best “serializing” of this time was the work done at the University of California at Berkeley by Michael Stonebraker. Stonebraker and his group were the first to address the fundamental principle of database replication and discuss a number of techniques that allowed for varying degrees of consistency. Many of these techniques rely on accessibility consistency—i.e., that is, to the user of the system it appears that there is only one system instead of a number of collaborating systems. Many systems during this time took the approach that it was better to fill the complete system than to break this transparency.

In the mid-1980s, with the rise of large Internet systems, these practices were switched. At that time people began to consider the idea that availability was a paramount concern, and it was not worth breaking the transparency. In this view, it was better to have a set of consistent replicas than it was to break the database into several replicas. This change in opinion was a major influence on the design of the Amazon system.
Problem

Poor guidelines on how to use the new programming models: are we weakening consistency too much, too little, just right?

“If no new updates are made to the database, then replicas will eventually reach a consistent state”
This particular example is a good one because, as we’ll see shortly, if there was a single overarching theme within the keynote talks, it turns out to be that strong synchronization of the sort provided by a locking service must be avoided like the plague. This doesn’t diminish the need for a tool like Chubby: when locking actually can’t be avoided, one wants a reliable, standard, provably correct
TOWARDS A CLOUD COMPUTING RESEARCH AGENDA

Ken Birman, Gregory Chockler, Robbert van Renesse

This particular example is a good one because, as we’ll see shortly, if there was a single overarching theme within the keynote talks, it turns out to be that strong synchronization of the sort provided by a locking service must be avoided like the plague. This doesn’t diminish the need for a tool like Chubby: when locking actually can’t be avoided, one wants a reliable, standard, provably correct

F1: A Distributed SQL Database That Scales

Jeff Shute
Chad Whipkey
David Menestrina
Radek Vingralek
Eric Rollins
Stephan Ellner
Traian Stancescu
Bart Samwel
Mircea Oancea
John Cieslewicz
Himani Apte
Ben Handy
Kyle Littlefield
Ian Rae*

Google, Inc.
*University of Wisconsin-Madison

ABSTRACT

F1 is a distributed relational database system built at Google to support the AdWords business. F1 is a hybrid database that combines high availability, the scalability of NoSQL systems like Bigtable, and the consistency and us-consistent and correct data. Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.
This particular example is a good one because, as we’ll see shortly, if there was a single overarching theme within the keynote talks, it turns out to be that strong synchronization of the sort provided by a locking service must be avoided like the plague. This doesn’t diminish the need for a tool like Chubby, and locking actually can’t be avoided—except, arguably, for rarely used, perfectly correct.

**Pay-off often worth it:** geo-distribution, offline access

**Theory can help:** guidelines and tools for relaxing synchronisation without compromising correctness

---

**ABSTRACT**

F1 is a distributed relational database system built at Google to support the AdWords business. F1 is a hybrid database that combines high availability, the scalability of NoSQL systems like Bigtable, and the consistency and use-consistent and correct data. Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.
Outline

- Replicated data types (CRDTs)
- Specification of consistency models
- Determining the right level of consistency
Outline

• Replicated data types (CRDTs)

• Specification of consistency models

• Determining the right level of consistency
Conflict resolution

cart = \{book\}

cart.add(\textit{book})

cart.remove(\textit{book})
Conflict resolution

cart = {book}

cart.add(\textit{book}) \quad \textbf{Conflict!} \quad \text{cart.remove(} \textit{book} \text{)}

Should the remove cancel the concurrent add?

Depends on application requirements
Conflict resolution

Remove wins:

Add wins:

Last writer wins:

choose based on operation time-stamps
Conflict resolution

```
cart = {book}
cart.add(book)  Conflict!  cart.remove(book)
```

- Most widespread application - collaborative editing: Google Docs, Office Online
- Low latency requires relaxing synchronisation, leads to conflicts
Replicated data types (CRDTs)

[Shapiro et al., SSS'11]

• Encapsulate conflict-resolution policies:

  Object ➔ Type ➔ Conflict resolution policy

• Not just read-write registers

• Examples: counter, register, set
Replicated data types

Replica states: $\sigma \in \text{State}$
Replicated data types

Replica states: $\sigma \in \text{State}$

Return value: $[\text{op}]_{\text{val}} \in \text{State} \rightarrow \text{Value}$
Replicated data types

Replica states: $\sigma \in \text{State}$

Return value: $\llbracket \text{op} \rrbracket_{\text{val}} \in \text{State} \rightarrow \text{Value}$

Effector: $\llbracket \text{op} \rrbracket_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Replicated data types

Replica states: $\sigma \in \text{State}$

Return value: $\llbracket \text{op} \rrbracket_{\text{val}} \in \text{State} \rightarrow \text{Value}$

Effector: $\llbracket \text{op} \rrbracket_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Replicated data types

Replica states: $\sigma \in \text{State}$

Return value: $\llbracket \text{op} \rrbracket_{\text{val}} \in \text{State} \rightarrow \text{Value}$

Effector: $\llbracket \text{op} \rrbracket_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Replicated data types

Replica states: $\sigma \in \text{State}$

Return value: $\llbracket \text{op} \rrbracket_{\text{val}} \in \text{State} \rightarrow \text{Value}$

Effector: $\llbracket \text{op} \rrbracket_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Counter

\[ \sigma \]

\[ [\text{op}]_{\text{val}} \]

\[ [\text{op}]_{\text{eff}}(\sigma) \]

\[ \sigma' \]

\[ [\text{op}]_{\text{eff}}(\sigma)(\sigma') \]

State = \( \mathbb{Z} \)

\[ [\text{read()}]_{\text{val}}(\sigma) = \sigma \]

\[ [\text{read()}]_{\text{eff}}(\sigma) = \lambda \sigma. \sigma \]
\[ \text{Counter} \]

\[ \sigma \]

\[ \text{val} \]

\[ \text{Counter} \]

\[ \sigma' \]

\[ \text{val} \]

\[ \text{add}(100) \]

\[ \lambda \sigma' . (\sigma' + 100) \]
Counter

$\sigma$

$\text{Counter}$

$\text{add}(100)_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100)$
\[ \text{\texttt{add(100)}}_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100) \]
\[ [\text{add}(100)]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma + 100) \]
\[
\text{add}(100) \quad \Rightarrow \quad \text{eff}(\sigma) = \lambda \sigma'. (\sigma + 100)
\]
$\left[\text{add}(100)\right]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma + 100)$
\[ \text{add}(100) \] \quad \text{eff}(\sigma) = \lambda \sigma'. (\sigma + 100) \\

Eventual consistency violated!
Ensuring eventual consistency

• **Effectors have to commute:**

\[
\forall \text{op}_1, \text{op}_2, \sigma_1, \sigma_2. \ [\text{op}_1]_{\text{eff}}(\sigma_1) \ ; \ [\text{op}_2]_{\text{eff}}(\sigma_2) = \ [\text{op}_2]_{\text{eff}}(\sigma_2) \ ; \ [\text{op}_1]_{\text{eff}}(\sigma_1)
\]

• **Eventual consistency:** replicas receiving the same messages in different orders end up in the same state
Last-writer-wins register

- Conflict arbitrated using timestamps
- Link to shared-memory consistency models
Last-writer-wins register

State  =  Value \times \text{Timestamp}

\[ [\text{read()}]_{\text{val}}(v, t) = v \]
Last-writer-wins register

\[ \text{write}(v_{\text{new}}) \text{ eff}(v, t) = \]
\[
\text{let } t_{\text{new}} = \text{newTS()} \text{ in }
\lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)\]
Last-writer-wins register

\[
\left[\text{write}(v_{\text{new}})\right]_{\text{eff}}(v, t) =
\]
\[
\text{let } t_{\text{new}} = \text{newTS()} \text{ in}
\]
\[
\lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)
\]
Last-writer-wins register

\[ [\text{write}(v_{\text{new}})]_{\text{eff}}(v, t) = \]

\[
\text{let } t_{\text{new}} = \text{newTS}() \text{ in }
\]

\[
\lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)
\]
Last-writer-wins register

\[ \text{\texttt{write}}(v_{\text{new}}) \text{eff}(v, t) = \]

let \( t_{\text{new}} = \text{newTS}() \) in

\( \lambda(v', t'). \text{if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t) \)
Last-writer-wins register

\[
\text{write}(1) \quad \text{write}(2)
\]

\[
\text{read}(): 2
\]

\[
\text{let } t_{\text{new}} = \text{newTS}() \text{ in }
\lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)
\]

\[
\left[\text{write}(v_{\text{new}})\right]_{\text{eff}}(v, t) =
\]
Last-writer-wins register

\[
\text{\texttt{write(1)}} \quad \text{\texttt{write(2)}}
\]

\[
\begin{align*}
\text{\texttt{read()}: 2} & \quad \text{\texttt{read()}: 2} \\
\end{align*}
\]

\[
\text{\texttt{\llbracket write(v_{new})\rrbracket_{\text{eff}}(v, t) =}} \\
\text{let } t_{new} = \text{newTS()} \text{ in} \\
\text{\texttt{\lambda(v', t'). if } t_{new} > t' \text{ then } (v_{new}, t_{new}) \text{ else } (v, t)}
\]
Last-writer-wins register

\[
\begin{align*}
\text{write}(1) & \quad t_1 < t_2 \\
\text{write}(2) &
\end{align*}
\]

\[
\begin{aligned}
\llbracket \text{write}(v_{\text{new}}) \rrbracket_{\text{eff}}(v, t) &= \\
&= \text{let } t_{\text{new}} = \text{newTS}() \text{ in} \\
&\quad \lambda(v', t'). \text{if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)
\end{aligned}
\]
Add-wins set

cart = {book}

cart.add(book)

cart = {book}

cart.remove(book)

cart = {book}
Add-wins set

- remove() acts differently wrt add() depending on whether it's concurrent or not
- Each addition creates a new instance:
  
  State = set of pairs (element, unique id)
Each `add()` creates a new element instance:

\[ \text{\texttt{add}}(x) \texttt{eff}(\sigma) = \lambda \sigma'. (\sigma' \cup \{(x, \text{uniqueid}())\}) \]
Each add() creates a new element instance:

\[
\llbracket \text{add}(x) \rrbracket_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' \cup \{(x, \text{uniqueid}())\})
\]
\{ (book, 1) \}

add(book)

\{ (book, 1), (book, 2) \}
\[
\text{add}(\text{book})
\]
\[
\{(\text{book}, 1)\}
\]
\[
\text{add}(\text{book})
\]
\[
\{(\text{book}, 1), (\text{book}, 2)\}
\]
\[
\text{read()} : \{\text{book}\}
\]

Instance ids ignored when reading the set:

\[
\llbracket \text{read()} \rrbracket_{\text{val}}(\sigma) = \{x \mid \exists \text{id.} \ (x, \text{id}) \in \sigma\}
\]
add(\{ \textit{book},1 \}\})

\textbf{add(\textit{book})}


{\textit{book},1}\}{

\{\textit{book},1}\}

\textbf{remove(\textit{book})}
remove(x) removes all currently present instances of x:

\[ \begin{align*}
\text{remove}(x) \text{ eff}(\sigma) & = \lambda \sigma'. (\sigma' \setminus \{(x, \text{id}) \in \sigma\}) \\
\end{align*} \]
remove(x) removes all currently present instances of x:

\[
\llbracket \text{remove}(x) \rrbracket_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' \setminus \{(x, \text{id}) \in \sigma\})
\]
remove\texttt{(x)} removes all currently present instances of \texttt{x}:\n
\[[\texttt{remove(x)}]_{\texttt{eff}}(\sigma) = \lambda \sigma'. (\sigma' \setminus \{(x, id) \in \sigma\})\]
add(\textit{book})

\{(\textit{book}, 1)\}


\{(\textit{book}, 2)\}

remove(\textit{book})

\{(\textit{book}, 1)\}

\emptyset
\[ \lambda \sigma'. \sigma' \cup \{(book, 2)\} \]

Effectors commutative \(\Rightarrow\) replicas converge
Outline

• Replicated data types (CRDTs)

• Specification of consistency models

• Determining the right level of consistency
Specification rationale

Conflict-resolution policies + anomalies

• Abstracts from implementation details

• Allows obtaining theoretical results: impossibility, inherent complexity

• Axiomatic style: connects to weak shared-memory models
Sequential data type semantics

Strong consistency $\rightarrow$ operations are totally ordered:

```
set.add(book)
```

```
set.remove(book)
```

```
set.read() : ∅
```

Compute the result by applying operations in sequence
Replicated data type semantics

set.add(book)

Delivered?

set.read() : ?

Only updates that have been delivered to the replica performing the operation are important
Replicated data type semantics

Abstract by the *visibility* relation on operations (acyclic, ...)

```
set.add(book)
```

Delivered?

Visible?

```
set.read() : ?
```
Abstract by the visibility relation on operations (acyclic, ...)

Replicated data type semantics

set.add(book)

Delivered?

set.remove(book) ➔

set.read() : ?

set.add(book)  set.remove(book)

vis

read() : ?

set.add(book)

set.remove(book)

set.read() : ?

Visible?
Replicated data type specification

\[ F: \text{context}(\text{op}) \rightarrow \text{return value}(\text{op}) \]

**Context:** all updates visible to the operation and the visibility relation between them + one more thing

```
                  /           /                 \\
               vis          vis              vis
                  \\
               set.read() : ?
```
Replicated data type specification

\[ F: \text{context}(\text{op}) \rightarrow \text{return value}(\text{op}) \]

**Context:** all updates visible to the operation and the visibility relation between them + one more thing

```
set.add(book)
set.add(book)
set.remove(book)
```

```
set.read() : ?
```
Replicated data type specification

\[ F: \text{context}(op) \rightarrow \text{return value}(op) \]

**Context:** all updates visible to the operation and the visibility relation between them + one more thing

```
set.add(book)
set.add(book)
set.remove(book)
set.read() : ?
```
Add-wins set

F: \texttt{context}(op) \rightarrow \texttt{return value}(op)

\textbf{Context:} all updates visible to the operation and the visibility relation between them + one more thing

\begin{align*}
\text{set.add}(\text{book}) & \quad \text{set.add}(\text{book}) & \quad \text{set.remove}(\text{book}) \\
\text{set.read() : ?} & \quad \text{vis} & \quad \text{vis} & \quad \text{vis}
\end{align*}
Add-wins set

F: $\text{context}(\text{op}) \rightarrow \text{return value}(\text{op})$

**Context:** all updates visible to the operation and the visibility relation between them + one more thing
Add-wins set

\[ F: \text{context}(\text{op}) \rightarrow \text{return value}(\text{op}) \]

**Context:** all updates visible to the operation and the visibility relation between them + one more thing

- `set.add(book)`
- `set.add(book)`
- `set.remove(book)`

- `set.read() : ?`
Add-wins set

\[ F: \text{context}(\text{op}) \rightarrow \text{return value}(\text{op}) \]

**Context:** all updates visible to the operation and the visibility relation between them + one more thing

```
set.add(book)
set.remove(book)
```

```
set.read() : \{book\}
```
Add-wins set

F: context(op) \rightarrow return value(op)

Context: all updates visible to the operation and the visibility relation between them + one more thing

\[\text{set.add}(book) \quad \text{set.add}(book) \quad \text{set.remove}(book)\]

\[\text{set.read}() : \{book\}\]

F: cancel all adds seen by a remove
Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + one more thing

F: cancel all adds seen by a remove
Last-writer-wins register

write(1) \[ t_1 \]
write(2) \[ t_2 \]

read(): 2

\[ t_1 < t_2 \]
Last-writer-wins register

write(1) \[ t_1 \]
\[ t_1 < t_2 \]
write(2)

read(): 2
Arbitrated before
read(): 2

Arbitration specifies extra information used to determine the outcome uniquely
Last-writer-wins register

F: context of op $\rightarrow$ return value of op

Context: all updates visible to the operation and the visibility and arbitration relations between them

\[
\text{reg.write}(1) \xrightarrow{ar} \text{reg.write}(2) \xrightarrow{ar} \text{reg.write}(3)
\]

\[
\text{reg.read}() : 3
\]

F: return the value of the last write in ar
Where do \texttt{vis} \\& \texttt{ar} come from?

Almost arbitrary: e.g., little control over when updates are visible to other replicas

\begin{tikzpicture}
  \node[draw, shape=cylinder, aspect=2,rotate=90] (db1) at (0,0) {};
  \node[draw, shape=cylinder, aspect=2,rotate=90] (db2) at (2,0) {};
  \node[draw, shape=ellipse, minimum height=1cm] (user) at (-1.5,-1) {};

  \draw[->] (user) -- node[above] {set.add(\textit{book})} (db1);

  \draw[red, ->] (db1) -- node[above] {?} (db2);
\end{tikzpicture}
Where do do *vis* & *ar* come from?

Almost arbitrary: e.g., little control over when updates are visible to other replicas.

```
set.add(book)  # 000 -- set.read() : {book}
```

But may guarantee that they don’t change unpredictably in the same `session` ➔ disallow anomalies.
Where do \textit{vis} & \textit{ar} come from?

Almost arbitrary: e.g., little control over when updates are visible to other replicas.

Almost arbitrary: e.g., little control over when updates are visible to other replicas.

But may guarantee that they don’t change unpredictably in the same \textit{session} \Rightarrow disallow anomalies.
Causal consistency

counter.add(100)

set.add(notification)

set.read() \ni \not notification

counter.read() : 0
Disallowed by causal consistency
Abstract execution: \((E, \text{so, vis, ar})\)

Generalises operation contexts

counter.add(100)

set.add(notification)

counter.read() : 0
Abstract execution: \((E, so, vis, ar)\)

Generalises operation contexts

All events that happened on the interface client/database & relations between them
Abstract execution: \((E, \text{so, vis, ar})\)

Operations grouped by clients and arranged in session order

Client 1

\[
\begin{align*}
\text{counter.add(100)} & \quad \text{session order} \\
\text{set.add(notification)} & \quad \text{vis}
\end{align*}
\]

Client 2

\[
\begin{align*}
\text{set.read() } & \ni \text{ notification} \\
\text{counter.read() } & : 0
\end{align*}
\]

Operations grouped by clients and arranged in session order
Abstract execution: (E, so, \text{vis}, ar)

Determines the context of every operation

Context(op) = \text{projection onto events visible to } op

return value(op) = F(\text{Context}(op))

counter.add(100)

set.add(notification)

set.read() \ni notification

counter.read() : 0
Abstract execution: \((E, \text{so}, \text{vis}, \text{ar})\)

Determines the context of every operation

\[
\begin{align*}
\text{counter.add(100)} & \quad \text{set.read()} \ni \text{notification} \\
\text{set.add(notification)} & \quad \text{counter.read()} : 100
\end{align*}
\]

\[
\text{Context}(\text{op}) = \text{projection onto events visible to } \text{op}
\]

\[
\text{return value}(\text{op}) = F(\text{Context}(\text{op}))
\]
Abstract execution: \((E, \text{so}, \text{vis}, \text{ar})\)

Determines the context of every operation

```
counter.add(100)
set.add(notification)
set.read() \ni \text{notification}
counter.read() : 100
```

Context\((\text{op})\) = projection onto events visible to \(\text{op}\)

return value\((\text{op})\) = F(Context(\text{op}))
Consistency axioms

Consistency axioms disallow anomalies by constraining executions

counter.add(100)  
set.add(notification)  
set.read() \ni \not notification

counter.read() : 0
Causal consistency: $(so \cup vis)^+ \subseteq vis$

$(so \cup vis \cup ar)$ is acyclic
Consistency axioms

Consistency axioms disallow anomalies by constraining executions

Causal consistency: \((so \cup vis)^+ \subseteq vis\)

\((so \cup vis \cup ar)\) is acyclic
Causal consistency:

**Principle:** strengthen consistency by mandating that more edges be included into `vis` and `ar`

Causal consistency: \((\text{so} \cup \text{vis})^+ \subseteq \text{vis}\)

\((\text{so} \cup \text{vis} \cup \text{ar})\) is acyclic
Basic eventual consistency

Session guarantees

Per-object causal consistency

Causal consistency

Strong consistency

Figure 1. Axioms of eventual consistency

--- WELL-FORMEDNESS AXIOMS ---

SOWF: so is the union of transitive, irreflexive and total orders on actions by each session

VISWF: $\forall a, b. a \xrightarrow{vis} b \implies \text{obj}(a) = \text{obj}(b)$

ARWF: $\forall a, b. a \xrightarrow{ar} b \implies \text{obj}(a) = \text{obj}(b)$,

ar is transitive and irreflexive, and

$\text{ar}_{\text{vis}-1}(a)$ is a total order for all $a \in A$

--- AUXILIARY RELATIONS ---

Per-object session order: $\text{soo} = (so \cap \text{sameobj})$

Per-object causality order: $\text{hbo} = (\text{soo} \cup \text{vis})^+$

Causality order: $\text{hb} = (so \cup \text{vis})^+$

--- BASIC EVENTUAL CONSISTENCY AXIOMS ---

RVAL: $\forall a \in A. \ \text{rval}(a) = \text{F}_{\text{type}(a)}(\text{cone}(a))$

EVENTUAL:

$\forall a \in A. \neg(\exists \text{ infinitely many } b \in A. \text{sameobj}(a, b) \land \neg(a \xrightarrow{vis} b))$

THINAIR: so $\cup$ vis is acyclic

--- SESSION GUARANTEES ---

RYW (Read Your Writes): soo $\subseteq$ vis

MR (Monotonic Reads): (vis; soo) $\subseteq$ vis

WFRV (Writes Follow Reads in Visibility): (vis; soo$^*$; vis) $\subseteq$ vis

WFRA (Writes Follow Reads in Arbitration): (vis; soo$^*$) $\subseteq$ ar

MWV (Monotonic Writes in Visibility): (soo; vis) $\subseteq$ vis

MWA (Monotonic Writes in Arbitration): soo $\subseteq$ ar

--- CAUSALITY AXIOMS ---

POCV (Per-Object Causal Visibility): hbo $\subseteq$ vis

POCA (Per-Object Causal Arbitration): hbo $\subseteq$ ar

COCV (Cross-Object Causal Visibility): (hb $\cap$ sameobj) $\subseteq$ vis

COCA (Cross-Object Causal Arbitration): hb $\cup$ ar is acyclic
Basic eventual consistency

- Processors and languages don’t provide strong consistency: weak memory models

Session guarantees

- Our specifications similar to weak memory model definitions

Per-object causal consistency

- Consistency axioms for registers $\approx$ C/C++ memory model

$\approx$ C/C++ relaxed

Causal consistency

$\approx$ C/C++ release/acquire

Strong consistency

```plaintext
RVAL: $\forall a \in A. \text{rval}(a) = F_{\text{type}(a)}(\text{cone}(a))$
EVENTUAL:
$\forall a \in A. \neg (\exists$ infinitely many $b \in A. \text{sameobj}(a, b) \land \neg (a \xrightarrow{\text{vis}} b))$
THINAIR: so $\cup$ vis is acyclic

SESSION GUARANTEES
RYW (Read Your Writes): soo $\subseteq$ vis
MR (Monotonic Reads): (vis; soo) $\subseteq$ vis
WFRV (Writes Follow Reads in Visibility): (vis; soo$^*$; vis) $\subseteq$ vis
WFRA (Writes Follow Reads in Arbitration): (vis; soo$^*$) $\subseteq$ ar
MWV (Monotonic Writes in Visibility): (soo; vis) $\subseteq$ vis
MWA (Monotonic Writes in Arbitration): soo $\subseteq$ ar

CAUSALITY AXIOMS
POCV (Per-Object Causal Visibility): hbo $\subseteq$ vis
POCA (Per-Object Causal Arbitration): hbo $\subseteq$ ar
COCV (Cross-Object Causal Visibility): (hb $\cap$ sameobj) $\subseteq$ vis
COCA (Cross-Object Causal Arbitration): hb $\cup$ ar is acyclic
```
Putting it all together

Conflict resolution policies ➔ Data type spec
Anomalies ➔ Consistency axioms

(E, so, vis, ar)
Putting it all together

Conflict resolution policies → Data type spec
Anomalies → Consistency axioms

(request₁, response₁, request₂, response₂, ...)

(E, so, vis, ar)
Putting it all together

Conflict resolution policies

Anomalies

➔ Data type spec

➔ Consistency axioms

Events \((E, so)\) allowed iff

\[ \exists \text{ execution } (E, so, vis, ar) \]

satisfying data type specs and axioms
Putting it all together

Conflict resolution policies

Anomalies

➔ Data type spec

➔ Consistency axioms

[POPL’14]

Events \((E, so)\) allowed iff

\(\exists\) execution \((E, so, vis, ar)\)

satisfying data type specs and axioms
Theoretical results

• Causal consistency is the strongest model that provides availability under network partitions [Attiya et al., PODC'15]

Terms and conditions apply
Theoretical results

• Causal consistency is the strongest model that provides availability under network partitions [Attiya et al., PODC'15]

Terms and conditions apply

• Some replicated data types have an inherently high metadata overhead: can't discard a character when deleting it from a Google Docs document! [Attiya et al., PODC'16]
Outline

• Replicated data types (CRDTs)

• Specification of consistency models

• Determining the right level of consistency
Application correctness

• Does weakening consistency preserve application correctness?

*Integrity invariants: account balance is non-negative, only registered students are enrolled into a course*

• Vanilla weak consistency often too weak to preserve correctness

• Need to strengthen consistency in parts of the application
Application correctness

- Does weakening consistency preserve application correctness?

  *Integrity invariants:* account balance is non-negative, only registered students are enrolled into a course

- Vanilla weak consistency often too weak to preserve correctness

- Need to strengthen consistency in parts of the application
\[
[\text{add}(100)]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100)
\]
Withdrawals

\[ \left[ \text{withdraw}(100) \right]_{\text{eff}}(\sigma) = \]
if \( \sigma \geq 100 \) then \( (\lambda \sigma'. \sigma' - 100) \) else \( (\lambda \sigma'. \sigma') \)
Withdrawals

\[
\text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma')
\]
Withdrawals

\[
\llbracket \text{withdraw}(100) \rrbracket_{\text{eff}}(\sigma) = \begin{cases} 
\lambda \sigma'. \sigma' - 100 & \text{if } \sigma \geq 100 \\
\lambda \sigma'. \sigma' & \text{else}
\end{cases}
\]
Withdrawals

\[ [[\text{withdraw}(100)]]_{\text{eff}}(\sigma) = \begin{cases} (\lambda \sigma'. \sigma' - 100) & \text{if } \sigma \geq 100 \\ (\lambda \sigma' \cdot \sigma') & \text{else} \end{cases} \]
\[
\lceil \text{withdraw}(100) \rceil_{\text{eff}}(\sigma) = \\
\text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma')
\]
balance = 100
withdraw(100) : ✔
balance = 100
withdraw(100) : ✔
balance = 0
balance = 0

λσ'. σ' - 100

⟦withdraw(100)⟧_{eff}(σ) =

if σ ≥ 100 then (λσ'. σ' - 100) else (λσ'. σ')
balance = 100
withdraw(100) : ✓
balance = 0
withdraw(100) : ✓
balance = 0
withdraw(100) : ✓
balance = -100

\[ [\text{withdraw}(100)]_{\text{eff}}(\sigma) = \]

\[
\begin{cases} 
\lambda \sigma'. \sigma' - 100 & \text{if } \sigma \geq 100 \\
\lambda \sigma'. \sigma' & \text{else}
\end{cases}
\]
balance = 100

withdraw(100) : ✔

balance = 0

balance = 0

λσ'. σ' - 100

withdraw(100) : ✔

balance = -100

balance = 0

balance = 100

add(100) : ✔
balance = 100
withdraw(100) : ✔
balance = 0

balance = 100
withdraw(100) : ✔
λσ'. σ' - 100
balance = 0

balance = 100
add(100) : ✔
balance = 100
balance = -100

Tune consistency:
• Withdrawals strongly consistent
• Deposits eventually consistent
Consistency choices

• **Databases with multiple consistency levels:**
  ▸ Commercial: Amazon DynamoDB, Microsoft DocumentDB
  ▸ Research: Li+ OSDI’12; Terry+ SOSP’13; Balegas+ EuroSys’15; Li+ USENIX ATC’18

• **Pay for stronger semantics** with latency, possible unavailability and money

• **Hard to figure out the minimum consistency** necessary to maintain correctness

• **Verification method and tool**
Strengthening consistency

- Baseline model: causal consistency
- Problem: withdrawals are causally independent
Strengthening consistency

- Symmetric conflict relation on ops: $\bowtie \subseteq \text{Op} \times \text{Op}$, e.g., withdraw $\bowtie$ withdraw

- Conflicting operations cannot be causally independent
Strengthening consistency

- Symmetric conflict relation on ops: $\bowtie \subseteq \text{Op} \times \text{Op}$, e.g., `withdraw $\bowtie$ withdraw`
- Conflicting operations cannot be causally independent
Strengthening consistency

- Symmetric conflict relation on ops: ⨿ ⊆ Op × Op, e.g., withdraw ⨿ withdraw
- Conflicting operations cannot be causally independent
Strengthening consistency

- Symmetric conflict relation on ops: $\bowtie \subseteq \text{Op} \times \text{Op}$, e.g., $\text{withdraw} \bowtie \text{withdraw}$

- Conflicting operations cannot be causally independent

- Implemented using consensus

- $\text{add}()$ doesn't need synchronisation
Strengthening consistency

Do we always have $I = (balance \geq 0)$?
Assume invariant holds

Check it’s preserved after executing op
Effect applied in a different state!
\[ \sigma \in I \]

\[[\text{op}]_{\text{eff}}(\sigma) \]

\[ [\text{op}]_{\text{eff}}(\sigma)(\sigma') \in I? \]

\[ [\text{op}]_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

\[ [\text{withdraw}(100)]_{\text{eff}}(\sigma) = \]

\[ \text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma') \]
\[ \text{Effector safety: } f(\sigma) \text{ preserves } I \text{ when executed in any state satisfying } P: \{ I \land P \} f(\sigma) \{ I \} \]
\[ \langle \text{op} \rangle_{\text{eff}}(\sigma)(\sigma') \in I ? \]

\[ \langle \text{op} \rangle_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety**: \( f(\sigma) \) preserves \( I \) when executed in any state satisfying \( P \): \( \{I \wedge P\} f(\sigma) \{I\} \)

\( \{\text{bal} \geq 0 \wedge \text{bal} \geq 100\} \text{ bal := bal-100 } \{\text{bal} \geq 0\} \)
\( \sigma \in I \)  

\[ \text{⟦op⟧}_{\text{eff}}(\sigma) \]  

\( \sigma' \)  

\[ \text{⟦op⟧}_{\text{eff}}(\sigma)(\sigma') \in I? \]

\[ \text{⟦op⟧}_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety:** \( f(\sigma) \) preserves \( I \) when executed in any state satisfying \( P \):  

\[ \{I \land P\} f(\sigma) \{I\} \]

\[ \{\text{bal} \geq 0 \land \text{bal} \geq 100\} \quad \text{bal} := \text{bal}-100 \quad \{\text{bal} \geq 0\} \]
\[\text{\([\text{op}]_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if ...} \]

1. **Effector safety:** \(f(\sigma)\) preserves \(I\) when executed in any state satisfying \(P\):  \(\{I \land P\} f(\sigma) \{I\}\)

\(\{\text{bal} \geq 0 \land \text{bal} \geq 100\} \quad \text{bal} := \text{bal}-100 \quad \{\text{bal} \geq 0\}\)
\[\sigma \in I\]

\[\text{\texttt{op}}\]

\[\llbracket \text{\texttt{op}} \rrbracket_{\text{eff}}(\sigma)\]

\[\sigma'\]

\[\llbracket \text{\texttt{op}} \rrbracket_{\text{eff}}(\sigma)(\sigma') \in I?\]

\[\llbracket \text{\texttt{op}} \rrbracket_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...}\]

1. **Effector safety:** \(f(\sigma)\) preserves \(I\) when executed in any state satisfying \(P\):

\[
\{I \land P\} \quad f(\sigma) \quad \{I\}
\]

\[
\{\text{bal} \geq 0 \land \text{bal} \geq 100\} \quad \text{bal} := \text{bal}-100 \quad \{\text{bal} \geq 0\}
\]
\[\left[ \text{op} \right]_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety:** \( f(\sigma) \) preserves \( I \) when executed in any state satisfying \( P \): \[ \{ I \land P \} f(\sigma) \{ I \} \]

\[ \{ \text{bal} \geq 0 \land \text{bal} \geq 100 \} \quad \text{bal} := \text{bal} - 100 \quad \{ \text{bal} \geq 0 \} \]
\[ [\text{op}]_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety**: \( f(\sigma) \) preserves \( I \) when executed in any state satisfying \( P \): \( \{I \land P\} f(\sigma) \{I\} \)

\[ \{\text{bal} \geq 0 \land \text{bal} \geq 100\} \text{ bal := bal-100 } \{\text{bal} \geq 0\} \]
\[ [\text{op}]_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety:** \( f(\sigma) \) preserves \( I \) when executed in any state satisfying \( P \): 
\[
\{ I \land P \} f(\sigma) \{ I \}
\]

\[
\{ \text{bal} \geq 0 \land \text{bal} \geq 100 \} \text{ bal := bal-100 } \{ \text{bal} \geq 0 \} 
\]
\[ [\text{op}]_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety:** \( f(\sigma) \) preserves \( I \) when executed in any state satisfying \( P \): \( \{I \land P\} f(\sigma) \{I\} \)

2. **Precondition stability:** \( P \) will hold when \( f(\sigma) \) is applied at any replica
$\sigma \in \mathcal{I}$

Diagram:

- Database 1
- Database 2
- $\sigma \in \mathcal{I}$
- $\text{op}$
- $[\text{op}]_{\text{eff}}(\sigma)$
- $\sigma'$

Question: $P(\sigma')?$
\[ \sigma \in I \]

**op**

\[ [\text{op}]_{\text{eff}}(\sigma) \]

\( \sigma' \)

\( \text{P}(\sigma')? \)

op’s causal dependencies
• Causal consistency $\Rightarrow$ receive op’s causal dependencies before receiving op
- Causal consistency $\rightarrow$ receive $\text{op}$’s causal dependencies before receiving $\text{op}$
- But can have additional effectors of operations concurrent with $\text{op}$: $f, g, \ldots$
- Effectors commute, so $\sigma' = (f; g; \ldots)(\sigma)$
• Causal consistency $\rightarrow$ receive op’s causal dependencies before receiving op

• But can have additional effectors of operations concurrent with op: $f, g, ...$

• Effectors commute, so $\sigma' = (f; g; ...)(\sigma)$
Precondition stability: $P$ is preserved by any effector $f$ of any operation: $\{P\} f \{P\}$
Precondition stability: \( P \) is preserved by any effector \( f \) of any operation: \( \{ P \} f \{ P \} \)

\( \{ \text{bal} \geq 100 \} \; \text{bal} := \text{bal} + 100 \; \{ \text{bal} \geq 100 \} \)
Precondition stability: $P$ is preserved by any effector $f$ of any operation: $\{P\} f \{P\}$

$\{\text{bal} \geq 100\}$ bal := bal+100 $\{\text{bal} \geq 100\}$
Precondition stability: $P$ is preserved by any effector $f$ of any operation: $\{P\} f \{P\}$

$\{\text{bal} \geq 100\}$  $\text{bal} := \text{bal} + 100$  $\{\text{bal} \geq 100\}$  ✔
Precondition stability: $P$ is preserved by any effector $f$ of any operation: $\{P\} f \{P\}$

\[
\begin{align*}
\{bal \geq 100\} & \quad \text{bal} := \text{bal} + 100 \quad \{bal \geq 100\} \quad \checkmark \\
\{bal \geq 100\} & \quad \text{bal} := \text{bal} - 100 \quad \{bal \geq 100\}
\end{align*}
\]
Precondition stability: $P$ is preserved by any effector $f$ of any operation: $\{P\} f \{P\}$

\[
\{\text{bal }\geq 100\} \quad \text{bal} := \text{bal}+100 \quad \{\text{bal }\geq 100\} \quad \checkmark
\]

\[
\{\text{bal }\geq 100\} \quad \text{bal} := \text{bal}-100 \quad \{\text{bal }\geq 100\}
\]
Precondition stability: $P$ is preserved by any effector $f$ of any operation: $\{P\} f \{P\}$

$\{bal \geq 100\} \quad bal := bal + 100 \quad \{bal \geq 100\} \quad \checkmark$

$\{bal \geq 100\} \quad bal := bal - 100 \quad \{bal \geq 100\} \quad \times$
Precondition stability: $P$ is preserved by any effector $f$ of any non-conflicting operation: $\{P\} f \{P\}$
Precondition stability: $P$ is preserved by any effector $f$ of any non-conflicting operation: $\{P\} f \{P\}$

withdraw $\Join$ withdraw; $\neg (\text{add} \Join \text{withdraw})$ 

$\sigma \in I$
σ ∈ I

\([op]_{eff}(\sigma)\)

withdraw is a causal dependency of op

Precondition stability: \(P\) is preserved by any effector \(f\) of any non-conflicting operation: \(\{P\} f \{P\}\)

withdraw \(\nabla\) withdraw; \(\neg (\text{add} \nabla \text{withdraw})\) ✔
Precondition stability: $P$ is preserved by any effector $f$ of any non-conflicting operation: $\{P\} f \{P\}$

withdraw ⋈ withdraw; $\neg(\text{add} \bowtie \text{withdraw})$ ✔
Precondition stability: $P$ is preserved by any effector $f$ of any non-conflicting operation: $\{P\} f \{P\}$

withdraw $\bowtie$ withdraw; $\neg$(add $\bowtie$ withdraw) $\checkmark$
CEC tool: Correct Eventual Consistency

• Automates the proof rule
• Discharges verification conditions using SMT
• Automatically suggests tokens
• Developed by Sreeja Nair (UPMC, Paris)

[POPL'16], https://github.com/LightKone/correct-eventual-consistency-tool
Conclusion

• Programming models for weak consistency are more advanced: replicated data types, transactions

• Axiomatic specifications can define a wide range of consistency models and replicated data types

• Basis for theoretical results about implementability and its cost

• Verification methods enable weakening consistency without compromising correctness