Reasoning about Eventual Consistency
and Replicated Data Types

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Joint work with Sebastian Burckhardt (Microsoft Research),
Hongseok Yang (Oxford) and Marek Zawirski (UPMC & INRIA)
Shared-memory concurrency
Distributed systems
Geo-replicated databases

- Every data centre stores a complete replica of data
- Purpose: fault tolerance, minimising latency
Geo-replicated databases

- Every data centre stores a complete replica of data
- Purpose: fault tolerance, minimising latency
Geo-replicated databases

- Every data centre stores a complete replica of data
- Purpose: fault tolerance, minimising latency

```python
cart.add(book)
```
Strong consistency

- Database behaves like a single replica
- Implementation: ensure replicas are in sync ➔ wait until other replicas get updated

```python
cart.add(book)
```
Strong consistency

- Database behaves like a single replica
- Implementation: ensure replicas are in sync → wait until other replicas get updated
- Problem: **high latency**, can’t tolerate network partitions
- CAP theorem: impossible to get all of strong Consistency, Availability, Partition-tolerance
• Problem: high latency, *can’t tolerate network partitions*

• CAP theorem: impossible to get all of strong Consistency, Availability, Partition-tolerance
• Problem: high latency, can’t tolerate network partitions

• CAP theorem: impossible to get all of strong Consistency, Availability, Partition-tolerance
• Problem: high latency, can’t tolerate network partitions

• **CAP** theorem: impossible to get all of strong Consistency, Availability, Partition-tolerance
Update your replica now, propagate to others later

Weak consistency: exhibits anomalies, similar to weak memory
Weak consistency

- Update your replica now, propagate to others later
- Weak consistency: exhibits anomalies, similar to weak memory
Conflicts

set = \{book\}

set.add(laptop)

set.remove(book)
Conflicts

\[ \text{set} = \{ \text{book} \} \]

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

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

\[
\{\text{set} = \{ \text{laptop} \}\} \quad \{\text{set} = \{ \text{laptop} \}\}
\]

Operations \textbf{commute} \rightarrow \text{replicas eventually consistent}
Conflicts

\[
\text{set} = \{\text{book}\}
\]

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

\[
\text{set}.\text{remove}(\text{book})
\]
set.add("book")  
Conflict!  
set.remove("book")

Should the remove cancel the concurrent add?  
Depends on application requirements
Conflicts

Remove wins:

Add wins:

Last writer wins: choose based on operation time-stamps
Remove wins: \[ \text{set} = \emptyset \]

Add wins: \[ \text{set} = \{ \text{book} \} \]

Last writer wins: choose based on operation time-stamps
Replicated data types

aka CRDTs, cloud types

Object ➔ Type ➔ Conflict resolution policy

• Many data types: registers, counters, graphs, sequences, file systems [Shapiro+ 2011]

• Nontrivial implementations
Eventually consistent databases
Eventually consistent databases

- Sophisticated programming interfaces
- Semantics poorly understood: how but not what
Perspective

**Long-term goal:** use formal techniques to

- define the semantics of eventually consistent databases
- develop tools for reasoning about their behaviour
- improve programmability and efficiency

**This talk:** specification & verification of replicated data types [POPL’14]
Sequential data type semantics

Strong consistency $\rightarrow$ operations are totally ordered:

- set.add($book$)
- set.remove($book$)
- set.read() : $\emptyset$

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

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(\textit{book})

Delivered?

Visible?

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

Delivered:

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

Visible?

```
set.add(book)  set.remove(book)
read() : ?

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 + some other things

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

\[
\text{set.read()} : ?
\]
Replicated data type specification

F: context(op) → return value(op)

**Context:** all updates visible to the operation and the visibility relation between them + some other things
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 + some other things

```
set.add(\text{book})
```

```
set.add(\text{book})
```

```
set.remove(\text{book})
```

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

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things

set.add(book)

set.add(book) → set.remove(book)

set.read() : ?
Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things


set.read() : ?

If you saw it, it’s not a conflict
Add-wins set

F: context(op) → return value(op)

Context: all updates visible to the operation and the visibility relation between them + some other things

```
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 + some other things

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

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

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

**Context:** all updates visible to the operation and the visibility relation between them + some other things

\[
\begin{align*}
\text{set.add}(\text{book}) & \quad \text{set.add}(\text{book})
\end{align*}
\]

\[
\begin{align*}
\text{set.remove}(\text{book}) & \quad \text{set.read}() : \{\text{book}\}
\end{align*}
\]

\[ F: \text{cancel all adds seen by a remove} \]
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 + some other things

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

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

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

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

**F:** cancel all adds seen by a remove
Almost arbitrary: little control over when updates are visible to other replicas/how timestamps are assigned.

Where does \texttt{vis} come from?

```python
set.add(\texttt{book}) \quad \text{----------}\quad ?
```
Where does \texttt{vis} come from?

Almost arbitrary: little control over when updates are visible to other replicas/how timestamps are assigned

But may guarantee that they don't change unpredictably between operations = anomalies disallowed

\begin{itemize}
\item \texttt{set.add(\textit{book})}
\item \texttt{set.read(): \{\textit{book}\}}
\end{itemize}

\textbf{Consistency axioms} \approx weak memory
Verifying data type implementations

Naive add-wins set implementation

Implementation challenge: remove behaves differently wrt different adds of the same element
\[
S = \{(\text{book}, 1)\}
\]

set.add(\text{book})

\[
S = \{(\text{book}, 1), (\text{book}, 2)\}
\]

- Each add creates a new element instance: (element, unique instance id)
$S = \{(book, 1)\}$

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

$S = \{(book, 1), (book, 2)\}$

```
set.read() : \{book\}
```

- Each add creates a new element instance: (element, unique instance id)
- Instance ids ignored when reading the set
- Remove should remove all currently present instances of `book` from $S$
$S = \{(book, 1)\}, T = \emptyset$

- `set.add(book)`

$S = \{(book, 1), (book, 2)\}, T = \emptyset$

- `set.remove(book)`

$S = \emptyset, T = \{(book, 1)\}$

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

- But maintain the set of tombstones $T$: element instances removed
- Remove moves all instances of $book$ in $S$ to $T$
$S = \{(book, 1)\}, T = \emptyset$

set.add($book$)

$S = \{(book, 1)\}, (book, 2)\}, T = \emptyset$

set.read() : $\{book\}$

$S = \emptyset, T = \{(book, 1)\}$

set.remove($book$)

State-based implementation: sends its state snapshot to other replicas
State-based implementation:
sends its state snapshot to other replicas
\[
S = \{(book, 1)\}, T = \emptyset
\]

- \(set.add(book)\)

\[
S = \{(book, 1), (book, 2)\}, T = \emptyset
\]

- \(set.remove(book)\)

\[
S = \emptyset, T = \{(book, 1)\}
\]

- \(set.read() : \{book\}\)

\[
S = \emptyset, T = \{(book, 1)\}
\]

- Ignore arriving instances that are in \(T\)
set.add(book)

S = {(book, 1)}, T = Ø

set.add(book)

S = {(book, 1), (book, 2)}, T = Ø

set.read() : {book}

S = ∅, T = {(book, 1)}

set.remove(book)

S = ∅, T = {(book, 1)}

S = {(book, 2)}, T = {(book, 1)}

• Ignore arriving instances that are in T
• Add new arriving instances to S
\[ S = \{(book, 1)\}, T = \emptyset \]

- set.add(book)

\[ S = \{(book, 1), (book, 2)\}, T = \emptyset \]

- set.read() : \{book\}

\[ S = \emptyset, T = \{(book, 1)\} \]

- set.remove(book)

\[ S = \emptyset, T = \{(book, 1)\} \]

- set.read() : \{book\}
\[ S = \{(\text{book}, 1)\}, T = \emptyset \]

set.add(\text{book})

\[ S = \{(\text{book}, 1), (\text{book}, 2)\}, T = \emptyset \]

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

\[ S = \emptyset, T = \{(\text{book}, 1)\} \]

set.remove(\text{book})

\[ S = \{(\text{book}, 2)\}, T = \{(\text{book}, 1)\} \]

\[ S, T \]

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

\[ \text{Impl} \models F ? \]
Justify by an abstract execution: like an operation context, but includes all operations in a database run.
Determines the context of every operation: projection onto visible events
So can check results against data type specs
Data type correctness:

• ∀ concrete execution of the implementation with any sequence of client operations

• ∃ corresponding abstract execution satisfying data type specifications
Data type correctness:

• ∀ concrete execution of the implementation with any sequence of client operations

• ∃ corresponding abstract execution satisfying data type specifications

• Requires reasoning about all replicas and interactions between them

• Want to modularise reasoning: construct the abstract execution from separate system configuration components
Data type correctness:

- ∀ concrete execution of the implementation with any sequence of client operations
- ∃ corresponding abstract execution satisfying data type specifications

- **Replication-aware simulations:** generalise simulation relations for abstract data types
- **Tell us how to construct a part of the abstract execution corresponding to a replica state or a message**
set.add(book)

S = {(book, 1)}, T = ∅

set.add(book)

S = {(book, 1)}, T = ∅

set.remove(book)

S = ∅, T = {(book, 1)}

S = {(book, 2)}, T = {(book, 1)}

set.read() : {book}

S, T

replica state: σ  events that led to this state: A
set.add(book)

S = {(book, 1)}, T = ∅

set.add(book)

S = {(book, 1), (book, 2)}, T = ∅

set.remove(book)

S = ∅, T = {(book, 1)}

set.read() : {book}

S = {(book, 2)}, T = {(book, 1)}

set.read() : {book}

S, T
Simulation for add-wins set

\[ (S, T) \]

Set S: \{ (book, 2), (laptop, 3) \}

Tombstones T: \{ (book, 1) \}
Simulation for add-wins set

Set $S$: \{(book,2), (laptop,3)\}

Tombstones $T$: \{(book,1)\}

Reverse-engineer possible histories from the state
Simulation for add-wins set

Set $S$: $\{(book, 2), (laptop, 3)\}$

Tombstones $T$: $\{(book, 1)\}$

$(elt, id) \in S \cup T \iff \text{add}(elt)^{id} \in A$
Simulation for add-wins set

Set $S$: $\{(book,2), (laptop,3)\}$

Tombstones $T$: $\{(book,1)\}$

$(elt, id) \in S \cup T \iff \text{add}(elt)^{id} \in A$

$(elt, id) \in T \rightarrow \text{remove}(elt) \iff \text{add}(elt)^{id}$

A

$\text{add}(book)^1 \text{ add}(laptop)^3$

$\text{add}(book)^2 \text{ remove}(book)$
Simulation for add-wins set

Set $S$: \{(book,2), (laptop,3)\}

Tombstones $T$: \{(book,1)\}

$$(\text{elt, id}) \in S \cup T \iff \text{add(elt)}^{id} \in A$$

$$(\text{elt, id}) \in T \quad \rightarrow \quad \text{remove(elt)} \leftarrow \text{add(elt)}^{id}$$

$$(\text{elt, id}) \in S \quad \rightarrow \quad \neg \text{add(elt)}^{id} \rightarrow \text{remove(elt)}$$
Proof obligation 1. Relations preserved during a system run
Proof obligation 1. Relations preserved during a system run.
System step: executing an operation

\[ \sigma \xrightarrow{\text{op}()} \sigma' \]

Proof obligation 1. Relations preserved during a system run
System step: executing an operation

\[ A \]

\[ \sigma \quad \xrightarrow{\text{op()} : \text{res}} \quad \sigma' \]

Proof obligation 1. Relations preserved during a system run
System step: executing an operation

\[ A \xrightarrow{\text{op}()} : \text{res} \xrightarrow{} A' \approx A + \{\text{op}\} \]

Proof obligation 1. Relations preserved during a system run
Proof obligation 2. Relations imply that the abstract execution satisfies the data type spec:
\[ \text{res} = F(\text{Context}_{A'}(\text{op})) \]
Proof obligation 2. Relations imply that the abstract execution satisfies the data type spec:
\[ \text{res} = F(\text{Context}_{A'}(\text{op})) \]

Modular: considers the state of a single replica.
System step: receiving a message

\[(\sigma, m) \xrightarrow{\text{receive}(m)} \sigma'\]

Proof obligation 1 again. Relations preserved during a system run
System step: receiving a message

\[(A, B) \quad \xleftrightarrow{} \quad (\sigma, m) \xrightarrow{\text{receive}(m)} \sigma'\]

Proof obligation 1 again. Relations preserved during a system run.
System step: receiving a message

Proof obligation 1 again. Relations preserved during a system run
System step: receiving a message

Good news: modular - consider the state of a single replica and a message

Bad news: modularity leads to incompleteness - loses required global information
A and B parts of the same abstract execution ➔ can be correlated by some invariants

Visibility can’t contradict on events common to A and B

Union of visibility relations in A and B itself a well-formed visibility relation ➔ acyclic

Simulation relations per-component ➔ don’t give this
Solution: 2-stage verification

1. Fix a class of data types implementations with similar messaging behaviour
   
   State-based: propagate information by sending full replica state

2. Prove key global invariants non-modularly

3. Unpleasant, but done once for the class
Solution: 2-stage verification

1. Fix a class of data types implementations with similar messaging behaviour.

   *State-based:* propagate information by sending full replica state

   - Prove key global invariants non-modularly
   - Unpleasant, but done once for the class

2. For any implementation within the class

   - Verify it modularly using replication-aware simulations while assuming the global invariants
Solution: 2-stage verification

1. Fix a class of data types implementations with similar messaging behaviour
   
   *State-based: propagate information by sending full replica state*

   - Prove key global invariants non-modularly
   - Unpleasant, but done once for the class

2. For any implementation within the class
   
   - Verify it modularly using replication-aware simulations while assuming the global invariants

Technical details in the paper
Conclusion

• First techniques for reasoning about replicated data types
  ▶ Specifying the intended semantics
  ▶ Verifying implementation correctness

• Only the first step
  ▶ Replicated data types only one system component
  ▶ More work needed even for them

• Put eventually consistent distributed systems onto the PL/verification agenda