Weak Consistency

Constantin Enea
Ecole Polytechnique
Replicated objects

Distributed systems
Replicated objects

Distributed systems

Conflicting concurrent updates: how are they observed on different replicas?

Adversarial environments: crashes, network partitions
Pessimistic Replication

Using consensus algorithms to agree on an order between conflicting concurrent updates

- strong consistency
- availability

CAP theorem [Gilbert et al.'02]: strong cons. + availability + partition tolerance is impossible
Optimistic Replication

Each update is applied on the **local** replica and propagated **asynchronously** to other replicas.

- **Strong consistency** is not available.
- **Availability** is ensured.

Replicas may store different versions of data: **weak consistency**.
Optimistic data replication
Optimistic data replication

Optimistic replication: replicas are allowed to diverge
- operations are applied immediately at the submission site
Optimistic data replication

Optimistic replication: replicas are allowed to diverge
– operations are applied immediately at the submission site
Optimistic data replication

Optimistic replication: replicas are allowed to diverge
– operations are applied immediately at the submission site
– in the background, sites exchange and apply remote operations
Optimistic data replication

Optimistic replication: replicas are allowed to diverge
- operations are applied immediately at the submission site
- in the background, sites exchange and apply remote operations
Optimistic data replication

Optimistic replication: replicas are allowed to diverge

– operations are applied immediately at the submission site
– in the background, sites exchange and apply remote operations
Concurrent operations
Concurrent operations
Concurrent operations

\{2, 3\} → add 3
\{2, 3\} → rem 3
Concurrent operations

Solving conflicts between concurrent operations
– speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations
– speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations
- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations
- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations
– speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

– speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

- speculate and eventually, roll-back, e.g., Google App Engine Datastore
- convergent conflict resolution, e.g., CRDTs [Shapiro et al.'11]
Concurrent operations

Solving conflicts between concurrent operations

– speculate and eventually, roll-back, e.g., Google App Engine Datastore
– convergent conflict resolution, e.g., CRDTs [Shapiro et al.'11]
Concurrent operations

Solving conflicts between concurrent operations

- speculate and eventually, roll-back, e.g., Google App Engine Datastore
- *convergent conflict resolution*, e.g., CRDTs [Shapiro et al.'11]

“add wins”

• Correct operations? Allowed level of consistency between replicas?
  – by CAP theorem [Gilbert et al.’02], achieving strong consistency (linearizability) is impossible
  – various correctness criteria: eventual consistency, causal consistency, etc
Example: Key-value map

```plaintext
struct Timestamp(number: nat; rid: nat);
function lessthan(Timestamp(n1,rid1), Timestamp(n2,rid2)) : boolean {
    return (n1 < n2) ∨ (n1 == n2 ∧ rid1 < rid2);
}

message Update(key: Key, val: Val, ts: Timestamp) : reliable

role Replica(rid: nat) {
    var localclock: nat;
    var store: pmap(Key, pair(Val,Timestamp));

    operation read(key: Key) {
        match store[key] with
        ⊥ → { return undef; }
        (val,ts) → { return val; }
    }

    operation write(key: Key, val: Val) {
        localclock++; // advance logical clock
        store[key] := (val,ts);
        send Update(key, val, Timestamp(localclock, rid));
        return ok;
    }

    receive Update(key, val, ts) {
        if (store[key] = ⊥ ∨ store[key].second.lessthan(ts))
            store[key] := (val, ts);
        if (ts.number > localclock) // keep up with time
            clock := ts.number;
```
Example: OR-Set

payload set $E$, set $T$

initial $\emptyset$, $\emptyset$

query contains (element $e$) : boolean $b$
  let $b = (\exists n : (e, n) \in E)$

query elements () : set $S$
  let $S = \{e|\exists n : (e, n) \in E\}$

update add (element $e$)
  prepare ($e$)
    let $n = \text{unique}()$
  effect ($e$, $n$)
    $E := E \cup \{(e, n)\} \setminus T$

update remove (element $e$)
  prepare ($e$)
    let $R = \{(e, n)|\exists n : (e, n) \in E\}$
  effect ($R$)
    $E := E \setminus R$
    $T := T \cup R$
Formal Definitions

- Histories
- Abstract Executions
- Operation Context
- Replicated Data Types
- Return-value Consistency

(Using the formal framework in “Principles of Eventual Consistency” by S. Burckhardt)
Histories

**Definition 3.1 (History).** A *history* is an event graph \((E, \text{op}, \text{rval}, \text{rb}, \text{ss})\) where

(h1) \(\text{op} : E \rightarrow \text{Operations}\) describes the operation of an event.

(h2) \(\text{rval} : E \rightarrow \text{Values} \cup \{\nabla\}\) describes the value returned by the operation, or the special symbol \(\nabla\) (\(\nabla \notin \text{Values}\)) to indicate that the operation never returns.

(h3) \(\text{rb}\) is a natural partial order on \(E\), the *returns-before* order.

(h4) \(\text{ss}\) is an equivalence relation on \(E\), the *same-session* relation.

**Definition 3.2 (Well-formed History).** A history \((E, \text{op}, \text{rval}, \text{rb}, \text{ss})\) is *well-formed* if

(h5) \(x \xrightarrow{\text{rb}} y\) implies \(\text{rval}(x) \neq \nabla\) for all \(x, y \in E\).

(h6) for all \(a, b, c, d \in E\): \((a \xrightarrow{\text{rb}} b \land c \xrightarrow{\text{rb}} d) \Rightarrow (a \xrightarrow{\text{rb}} d \lor c \xrightarrow{\text{rb}} b)\).

(h7) For each session \([e] \in E/\approx_{\text{ss}}\), the restriction \(\text{rb}|_{[e]}\) is an enumeration.
Abstract Executions

Definition 3.3 (Abstract Executions). An abstract execution is an event graph \((E, op, rval, rb, ss, vis, ar)\) such that

(a1) \((E, op, rval, rb, ss)\) is a history.
(a2) \(vis\) is an acyclic and natural relation.
(a3) \(ar\) is a total order.

Definition 4.4 (Operation Context). An operation context is a finite event graph \(C = (E, op, vis, ar)\) where \(op : E \rightarrow Operations\) describes the operation of each event, \(vis\) is an acyclic relation representing visibility among the elements of \(E\), and \(ar\) is a total order representing arbitration of the elements in \(E\). We let \(C\) be the set of all operation contexts.
Replicated Data Types

**Definition 4.5** (Replicated Data Type). A *replicated data type* \( \mathcal{F} \) is a function \( \text{Operations} \times C \rightarrow \text{Values} \) that, given an operation \( o \) and an operation context \( C \), specifies the expected return value \( \mathcal{F}(o, C) \) to be used when performing \( o \) in context \( C \), and which does not depend on the events, *i.e.* is the same for isomorphic (as in Definition 2.2) contexts: \( C \approx C' \Rightarrow \mathcal{F}(o, C) = \mathcal{F}(o, C') \) for all \( o, C, C' \).

**Replicated Counter**: a read returns the number of increment operations in the context

\[
\mathcal{F}_{\text{ctr}}(\text{rd}, (E, \text{op}, \text{vis}, \text{ar})) = |\{ e' \in E \mid \text{op}(e') = \text{inc} \}|
\]

**Last-Writer-Wins Register**: a read returns the value of the last write in the context (w.r.t. arbitration order), or “undef”, if there is no write

\[
\mathcal{F}_{\text{reg}}(\text{rd}, (E, \text{op}, \text{vis}, \text{ar})) = \begin{cases} 
\text{undef} & \text{if } \text{writes}(E) = \emptyset \\
v & \text{if } \text{op}(\max_{\text{ar}} \text{writes}(E)) = \text{wr}(v) 
\end{cases}
\]

**Multi-Value Register**: a read returns a set of values, one for each write in the context that has not been overwritten by some other write

\[
\mathcal{F}_{\text{mvr}}(\text{rd}, (E, \text{op}, \text{vis}, \text{ar})) = \\
\{ v \mid \exists e \in E : \text{op}(e) = \text{wr}(v) \text{ and } \forall e' \in \text{writes}(E) : e \not\xrightarrow{\text{vis}} e' \}
\]
payload set $E$, set $T$

initial $\emptyset, \emptyset$
query contains (element $e$) : boolean $b$
  let $b = (\exists n : (e, n) \in E)$
query elements () : set $S$
  let $S = \{ e | \exists n : (e, n) \in E \}$
update add (element $e$)
  prepare ($e$)
    let $n = \text{unique()}$
effect ($e, n$)
  $E := E \cup \{(e, n)\} \setminus T$
update remove (element $e$)
  prepare ($e$)
    let $R = \{(e, n) | \exists n : (e, n) \in E \}$
effect ($R$)
  $E := E \setminus R$
  $T := T \cup R$

These CRDT specifications follow a new notation with mixed state- and operation-based update propagation. Although the formalization of this mixed model, and the associated proof obligations that check compliance to CRDT requisites, is out of the scope of this report the notation is easy to infer from the standard CRDT model [9, 8, 10].

System model synopsis:
We consider a single object, replicated at a given set of processes/replicas. A client of the object may invoke an operation at some replica of its choice, which is called the source of the operation. A query executes entirely at the source. An update applies its side effects first to the source replica, then (eventually) at all replicas, in the downstream for that update. To this effect, an update is modeled as an update pair ($p, u$) that includes two operations such that $p$ is a side-effect free prepare(-update) operation and $u$ is an effect(-update) operation; the source executes the prepare and effect atomically; downstream replicas execute only the effect $u$. In the mixed state- and operation-based modelling, replica state can both be changed by applying an effect operation or by merging state from another replica of the same object. The monotonic evolution of replica states is described by a compare operation, supplied with each CRDT specification.

4.1 Observed Remove Set
Figure 2 shows our specification for an add-wins replicated set CRDT. Its concurrent specification {$P$}$_{u_0} \not\subseteq \ldots \not\subseteq$ is for each element $e$ defined as follows:
Return-Value Consistency

Definition 4.8. For a replicated data type \( F \), we define the return value consistency guarantee as

\[
\text{RVal}(F) \overset{\text{def}}{=} \forall e \in E : \text{rval}(e) = F(\text{op}(e), \text{context}(A, e))
\]

where context is defined as follows:

Definition 4.9. Let \( A = (E, \text{op}, \text{rval}, \text{rb}, \text{ss}, \text{vis}, \text{ar}) \) be an abstract execution containing an event \( e \in E \). Then

\[
\text{context}(A, e) \overset{\text{def}}{=} A|_{\text{vis}^{-1}(e), \text{op}, \text{vis}, \text{ar}}
\]
Formal Definitions

**ReadMyWrites** \(\overset{\text{def}}{=} (so \subseteq vis)\)

**MonotonicReads** \(\overset{\text{def}}{=} (vis; so) \subseteq vis\)

**ConsistentPrefix** \(\overset{\text{def}}{=} (ar; (vis \cap \neg ss)) \subseteq vis\)

**NoCircularCausality** \(\overset{\text{def}}{=} \text{acyclic}(hb)\)

**CausalVisibility** \(\overset{\text{def}}{=} (hb \subseteq vis)\)

**CausalArbitration** \(\overset{\text{def}}{=} (hb \subseteq ar)\)

**Causality** \(\overset{\text{def}}{=} \text{CausalVisibility} \land \text{CausalArbitration}\)

**SingleOrder** \(\overset{\text{def}}{=} \exists E' \subseteq \text{rval}^{-1}(\nabla): \text{vis} = ar \setminus (E' \times E)\)

**Realtime** \(\overset{\text{def}}{=} rb \subseteq ar\)

**Linearizability** \((\mathcal{F})\) \(\overset{\text{def}}{=} \text{SingleOrder} \land \text{Realtime} \land \text{RVal}(\mathcal{F})\)

**SequentialConsistency** \((\mathcal{F})\) \(\overset{\text{def}}{=} \text{SingleOrder} \land \text{ReadMyWrites} \land \text{RVal}(\mathcal{F})\)

**CausalConsistency** \((\mathcal{F})\) \(\overset{\text{def}}{=} \text{EventualVisibility} \land \text{Causality} \land \text{RVal}(\mathcal{F})\)

**BasicEventualConsistency** \((\mathcal{F})\) \(\overset{\text{def}}{=} \text{EventualVisibility} \land \text{NoCircularCausality} \land \text{RVal}(\mathcal{F})\)

**EventualVisibility**: the nb. of operations not “seeing” an operation is finite
Causal Consistency (CC)  

If an update is visible, then all the updates is dependent on should be also visible.

- \texttt{write(x,1)} and \texttt{write(y,1)} are causally dependent:  

\begin{itemize}
  \item \texttt{write(x,1)}
  \item \texttt{write(y,1)}
\end{itemize}
Causal Consistency (CC)

If an update is visible, then all the updates it is dependent on should be also visible.

- \text{write}(x,1) \text{ and write}(y,1) \text{ are causally dependent}:
If an update is visible, then all the updates is dependent on should be also visible

- **write**(x,1) and **write**(y,1) are **concurrent**:
Formalization: Visibility

A is visible to operation B at replica R if the effect of A had been included in R at the time when B was executed.

\[
\text{write}(x,1) \quad \text{read}(x): 1 \quad \text{read}(y): 1
\]

\[
\downarrow \quad \text{po} \quad \downarrow \quad \text{po}
\]

\[
\text{write}(y,1) \quad \text{read}(x): 1
\]
A is visible to operation B at replica R if the effect of A had been included in R at the time when B was executed.
Formalization: Visibility

[Burckhardt et al.’14]

A is visible to operation B at replica R if the effect of A had been included in R at the time when B was executed.

∀ read. “return value is consistent with the set of visible ops”
Formalization: Visibility

[Burckhardt et al.’14]

A is visible to operation B at replica R if the effect of A had been included in R at the time when B was executed.

∃vis. vis ⊇ po ∧ vis transitive
∧ ∀read. “return value is consistent with the set of visible ops”
Formalization: Arbitration

[Burckhardt et al.'14]

Arbitration: conflict resolution between concurrent writes using timestamps

write(x,1)  write(x): 2
  po
  read(x): 2  po
  read(x): 2
Arbitration: conflict resolution between concurrent writes using timestamps
Formalization: Arbitration

Arbitration: conflict resolution between concurrent writes using timestamps
Formalization: Arbitration

Arbitration: conflict resolution between concurrent writes using timestamps

$∃ \text{vis, arb. arb} \ni \text{vis} \ni \text{po} \land \text{vis transitive} \land \text{arb total order}$

$\land \forall \text{read. "return value = value of the last visible write in arbitration order"}$
Arbitration: conflict resolution between concurrent writes using timestamps

∃ vis, arb. arb ⊇ vis ⊇ po ∧ vis transitive ∧ arb total order
∧ ∀ read. "return value = value of the last visible write in arbitration order"
Checking CC

Checking CC for a single execution is NP-complete (proof on whiteboard)

Checking CC for a finite-state implementation (given as a regular language) and a regular specification is undecidable

[POPL’17]
Characterizing CC

∃ vis, arb. arb ⊇ vis ⊇ po ∧ vis transitive ∧ arb total order
∧ ∀ read. “return value = value of the last visible write in arbitration order”
is equivalent to

∃ rf. po ∪ rf is acyclic and ( po ∪ rf ); rb is acyclic and po ∪ rf ∪ cf is acyclic

rf (read-from): relating each read to a write with the same variable/value
rb (read-before): relating each read to a write that overwrites the read value
cf (conflict): a necessary subset of arb derived from po and rf

read(x): _ ─_________→ write(x, _)

write(x, _) ┌─┐ ┌─┐
│ rf │ │ po ∪ rf │
Characterizing CC

∃ vis, arb. arb ⊇ vis ⊇ po ∧ vis transitive ∧ arb total order
∧ ∀ read. “return value = value of the last visible write in arbitration order”

is equivalent to

∃ rf. po ∪ rf is acyclic and ( po ∪ rf ); rb is acyclic and po ∪ rf ∪ cf is acyclic

rf (read-from): relating each read to a write with the same variable/value
rb (read-before): relating each read to a write that overwrites the read value
cf (conflict): a necessary subset of arb derived from po and rf
Characterizing CC

∃ vis, arb. arb ⊇ vis ⊇ po ∧ vis transitive ∧ arb total order
∧ ∀ read. “return value = value of the last visible write in arbitration order”

is equivalent to

∃ rf. po ∪ rf is acyclic and ( po ∪ rf ); rb is acyclic and po ∪ rf ∪ cf is acyclic

- **rf (read-from):** relating each read to a write with the same variable/value
- **rb (read-before):** relating each read to a write that overwrites the read value
- **cf (conflict):** a necessary subset of arb derived from po and rf
Characterizing CC

\[ CC \equiv \exists \text{rf. } \ldots \text{ and } \text{po} \cup \text{rf} \cup \text{cf} \text{ is acyclic} \]

Example:

- write(x, 1)
  - po
  - rf
  - read(x): 2
- write(x): 2
  - po
  - rf
  - read(x): 1

write(x, _) → cf → write(x, _)

write(x, _) → rf

read(x): _
Characterizing CC

\( CC = \exists rf. \ldots \text{ and} \)
\[ po \cup rf \cup cf \text{ is acyclic} \]

Example:
\( \text{write}(x,1) \xrightarrow{cf} \text{write}(x): 2 \)
\[ \text{read}(x): 2 \quad \text{read}(x): 1 \]

[POPL’17]
Characterizing CC

$CC = \exists \text{rf. } \ldots \text{ and } \text{po} \cup \text{rf} \cup \text{cf}$ is acyclic

Example:

- write(x, 1) → write(x): 2
- write(x, _) → write(x, _)
- read(x): _

$CC \equiv \exists \text{rf. } \ldots \text{ and } \text{po} \cup \text{rf} \cup \text{cf}$ is acyclic
Characterizing CC

∃ vis, arb. arb ⊇ vis ⊇ po ∧ vis transitive ∧ arb total order
∧ ∀ read. "return value = value of the last visible write in arbitration order"

is equivalent to

∃ rf. po ∪ rf is acyclic and ( po ∪ rf ); rb is acyclic and po ∪ rf ∪ cf is acyclic

Proof. (⇒) rf is included in vis (return-value consistency). vis is acyclic => po ∪ rf is acyclic.
( po ∪ rf ); rb cyclic => write(x, b) visible to read(x) => read(x) does not read last in arb ⊇ po ∪ rf
cf is included in arb (return-value consistency). po ∪ rf is included in vis and thus arb. arb acyclic
=> po ∪ rf ∪ cf acyclic.
Characterizing CC

∃ vis, arb. arb ⩾ vis ⩾ po ∧ vis transitive ∧ arb total order

∧ ∀ read. “return value = value of the last visible write in arbitration order”

is equivalent to

∃ rf. po ∪ rf is acyclic and (po ∪ rf); rb is acyclic and po ∪ rf ∪ cf is acyclic

Proof. (⇐) take vis = po ∪ rf and arb = any total order extension of po ∪ rf ∪ cf
Checking CC

\[ \text{CC} \equiv \exists \text{rf. } \text{po} \cup \text{rf} \text{ is acyclic and } (\text{po} \cup \text{rf}); \text{rb} \text{ is acyclic and } \text{po} \cup \text{rf} \cup \text{cf} \text{ is acyclic} \]

Each value is written once (data independence) => fixed read-from

Testing: acyclicity can be checked in polynomial time

Verification: using finite-state automata to represent “cyclic” executions