Reasoning about data consistency in 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]
• 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
Reasoning about data consistency in distributed systems

Alexey Gotsman

IMDEA Software Institute, Madrid, Spain

- Common application: collaborative editing (Google Docs, Office Online)
- Would accept edits before communicating with Google servers or other clients
NoSQL data stores

New generation of data stores with high scalability and low latency, but weak consistency

Cassandra
Amazon DynamoDB
Microsoft Azure DocumentDB
Riak

So what consistency guarantees do they provide?
Anomalies

access.write(noboss)

post.write(photo)
Anomalies

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all
Anomalies

Causal dependency: one operation is aware of another

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all
Anomalies

access.write(noboss)

post.write(photo)

Causal consistency model: disallows this anomaly

post.read() : photo

access.read() : all
Early days

Poor guidelines on how to use the weakly consistent data stores: are we weakening consistency too much, too little, just right?
Early days

Poor guidelines on how to use the weakly consistent data stores: 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”
Early days

Poor guidelines on how to use the weakly consistent data stores: 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”
Early days

Poor guidelines on how to use the weakly consistent data stores: 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.
Strong vs weak consistency

• Pay-off from weakening consistency often worth it: higher scalability, lower latency in geo-distribution, offline access
  ▶ Both strong and weak systems used in industry

• But programmers need help in using it:
  ▶ Programming abstractions for weak consistency
  ▶ Methods for reasoning about how weakening consistency affects application correctness
Also centralised SQL databases

Don't provide strong consistency either by default or at all: to exploit single-node concurrency
Also centralised SQL databases

Don't provide strong consistency either by default or at all: to exploit single-node concurrency

Granularity of Locks and Degrees of Consistency in a Shared Data Base

J.N. Gray
R.A. Lorie
G.R. Putzolu
I.L. Traiger

IBM Research Laboratory
San Jose, California

ABSTRACT: In the first part of the paper the problem of choosing the granularity (size) of lockable objects is introduced and the related tradeoff between concurrency and overhead is discussed. A locking protocol which allows simultaneous locking at various granularities by different transactions is presented. It is based on the introduction of additional lock modes besides the
Also centralised SQL databases

Don't provide strong consistency either by default or at all: to exploit single-node concurrency

Granularity of Locks and Degrees of Consistency in a Shared Data Base

J.N. Gray
R.A. Lorie
G.R. Putzolu
I.L. Traiger

IBM Research Laboratory
San Jose, California

ABSTRACT: In the first part of the paper, the granularity (size) of lockable objects is distinguished, and a related tradeoff between concurrency and locking protocol which allows small granularities by different transactions on the introduction of additional lock modes besides the

Are applications OK with this?
ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT
In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an ACIDRain attack, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and deployed on over 2M websites. We identify and verify 22 critical ACIDRain attacks that allow attackers to corrupt store inventory, over-spend gift cards, and steal inventory.

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the withdraw function could both read balance $100, check that $100 ≥ $99, and each allow $99 to be withdrawn, resulting $198 total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as SELECT FOR UPDATE is used. While this scenario closely resembles textbook examples of improper transaction use, in this paper, we show that widely-deployed eCommerce applications are similarly vulnerable to such ACIDRain attacks, allowing corruption of application state and theft of assets.

```python
1 def withdraw(amt, user_id):
2     bal = readBalance(user_id)
3     if (bal >= amt):
4         writeBalance(bal - amt, user_id)
```

```python
1 def withdraw(amt, user_id):
2     beginTxn()
3     bal = readBalance(user_id)
4     if (bal >= amt):
5         writeBalance(bal - amt, user_id)
6     commit()
```
ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an ACIDRain attack, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation and introduce a novel latent potential for non-serializable behavior under concurrency anomalies. We introduce a language-agnostic method for identifying potential isolation anomalies in web applications, called Anomaly Detection (2AD), that uses dynamic traces of application behavior and access patterns to reason about the space of possible concurrency anomalies. We apply a prototype 2AD analysis tool to a large-scale dataset of self-hosted eCommerce applications written in four languages and deployed on over 2M websites. We identify and verify 2 AD anomalies that allow attackers to corrupt application state, over-spend gift cards, and steal inventory.

```python
def withdraw(amt, user_id):  
    bal = readBalance(user_id)  
    if (bal >= amt):  
        writeBalance(bal - amt, user_id)
```

No! E-commerce applications can be hacked by exploiting weak consistency of back-end databases
Weak shared-memory models

- Multicore processors: x86, ARM
  
  *Multiprocessor ~ distributed system*

- Programming languages: C/C++, Java
  
  *Due to compiler optimisations*
This course

• Programming abstractions for weak consistency

• Methods for specification

• Methods and tools for reasoning about application correctness and consistency needs

• Implementing strong consistency
Strong consistency and the CAP theorem
• Database system manages a set of objects: \( \text{Obj} = \{x, y, z\ldots\} \)

• Objects associated with types \( \text{Type} = \{\tau, \ldots\} \)

• For each type \( \tau \in \text{Type} \):
  ▸ Set of operations \( \text{Op}_\tau \), including arguments
  ▸ Return values: \( \text{Val}_\tau \)
Data model

- Integer register
  - \( \text{Op}_{\text{intreg}} = \{\text{read}, \text{write}(k) \mid k \in \mathbb{Z}\} \)
  - \( \text{Val}_{\text{intreg}} = \mathbb{Z} \cup \{\text{ok}\} \)

- Counter:
  - \( \text{Op}_{\text{counter}} = \{\text{read}, \text{add}(k) \mid k \in \mathbb{N}\} \)
  - \( \text{Val}_{\text{counter}} = \mathbb{N} \cup \{\text{ok}\} \)
Sequential semantics

• Semantics in an ordinary programming language

• For each type $\tau \in \text{Type}$: set of states $\text{State}_\tau$, initial state $\sigma_0 \in \text{State}_\tau$

  ▸ $\text{State}_{\text{intreg}} = \mathbb{Z}$

  ▸ $\text{State}_{\text{counter}} = \mathbb{N}$

• Semantics of an operation $\text{op}$:

  ▸ $[\text{op}]_{\text{val}} \in \text{State}_\tau \rightarrow \text{Value}_\tau$

  ▸ $[\text{op}]_{\text{state}} \in \text{State}_\tau \rightarrow \text{State}_\tau$
Register semantics

- State = $\mathbb{Z}$
- $\llbracket \text{write}(k) \rrbracket_{\text{state}}(\sigma) = k$
- $\llbracket \text{write} \rrbracket_{\text{val}}(\sigma) = \text{ok}$
- $\llbracket \text{read} \rrbracket_{\text{state}}(\sigma) = \sigma$
- $\llbracket \text{read} \rrbracket_{\text{val}}(\sigma) = \sigma$
Counter semantics

- **State** = \( \mathbb{N} \)
- \( \llbracket \text{add}(k) \rrbracket_{\text{state}}(\sigma) = \sigma + k \)
- \( \llbracket \text{add}(k) \rrbracket_{\text{val}}(\sigma) = \text{ok} \)
- \( \llbracket \text{read} \rrbracket_{\text{state}}(\sigma) = \sigma \)
- \( \llbracket \text{read} \rrbracket_{\text{val}}(\sigma) = \sigma \)
Counter semantics

- State = $\mathbb{N}$
- $\sem{\text{add}(k)}_{\text{state}}(\sigma) = \sigma + k$
- $\sem{\text{add}(k)}_{\text{val}}(\sigma) = \text{ok}$
- $\sem{\text{read}}_{\text{state}}(\sigma) = \sigma$
- $\sem{\text{read}}_{\text{val}}(\sigma) = \sigma$

read-only operation:
$\sem{\text{op}}_{\text{state}}(\sigma) = \sigma$
Counter semantics

• State = $\mathbb{N}$

• $\llbracket \text{add}(k) \rrbracket_{\text{state}}(\sigma) = \sigma + k$
  update operation

• $\llbracket \text{add}(k) \rrbracket_{\text{val}}(\sigma) = \text{ok}$

• $\llbracket \text{read} \rrbracket_{\text{state}}(\sigma) = \sigma$
  read-only operation: $\llbracket \text{op} \rrbracket_{\text{state}}(\sigma) = \sigma$

• $\llbracket \text{read} \rrbracket_{\text{val}}(\sigma) = \sigma$
Clients issue requests and get responses: history records the interactions in a single execution.
Consistency specification

Assume every request yields a response
No next request until the previous one responded
Consistency specification

Assume every request yields a response
No next request until the previous one responded
Assume every request yields a response
No next request until the previous one responded
Consistency specification

request₁
response₁
request₂
response₂
...

request₁
response₁
request₂
response₂
...

event e
Consistency specification

Session order \( so \): the order in which events are issued: union of total per-client total orders
Consistency specification

Session order \( so \): the order in which events are issued:
union of total per-client total orders
Consistency specification

History $H = (E, so)$
Consistency specification

History $H = (E, so)$

Consistency model - a set of histories $\mathcal{H}$: the set of allowed database behaviours
Visualising histories

```
x.read: 0
  ↓ so
y.write(1)
  ↓ so
z.write(2)
  ↓ so
c.add(1)
  ↓ so
c.add(1)
```

```
x.write(1)
  ↓ so
c.add(1)
  ↓ so
c.read: 1
  ↓ so
z.read: 2
```
Visualising histories

x.read: 0
  ↓ so
y.write(1)
  ↓ so
z.write(2)
  ↓ so
c.add(1)
  ↓ so
c.add(1)

x.write(1)
  ↓ so
c.add(1)
  ↓ so
c.read: 1
  ↓ so
z.read: 2
Using a consistency model

- Consistency model $\mathcal{H}$: behaviour of the database under arbitrary clients
Using a consistency model

• Consistency model $\mathcal{H}$: behaviour of the database under arbitrary clients

• Program $P \rightarrow$ set of all executions $[P]$ under arbitrary behaviour of the database
Using a consistency model

- Consistency model $\mathcal{H}$: behaviour of the database under arbitrary clients

- Program $P \rightarrow$ set of all executions $\llbracket P \rrbracket$ under arbitrary behaviour of the database

- Semantics of $P$ when using $\mathcal{H}$:
  \[
  \llbracket P, \mathcal{H} \rrbracket = \{X \in \llbracket P \rrbracket \mid \text{history}(X) \in \mathcal{H}\}
  \]
Using a consistency model

• Consistency model $\mathcal{H}$: behaviour of the database under arbitrary clients

• Program $P \rightarrow$ set of all executions $⟦P⟧$ under arbitrary behaviour of the database

• Semantics of $P$ when using $\mathcal{H}$:

  $⟦P, \mathcal{H}⟧ = \{X \in ⟦P⟧ | \text{history}(X) \in \mathcal{H}\}$

$P$:
$r1 = x.read();$
$r2 = x.read();$
y.write(r1==r2);
Using a consistency model

- Consistency model $\mathcal{H}$: behaviour of the database under arbitrary clients
- Program $P \rightarrow$ set of all executions $[P]$ under arbitrary behaviour of the database
- Semantics of $P$ when using $\mathcal{H}$:
  $[P, \mathcal{H}] = \{X \in [P] \mid \text{history}(X) \in \mathcal{H}\}$

$P:$
- $r1 = x.\text{read}();$
- $r2 = x.\text{read}();$
- $y.\text{write}(r1==r2);$

$[P]:$
- $x.\text{read}(): 42;$
- $x.\text{read}(): 42;$
- $x.\text{read}(): 43;$
- $y.\text{write}(1);$
- $y.\text{write}(0);$
Using a consistency model

• Consistency model $\mathcal{H}$: behaviour of the database under arbitrary clients

• Program $P \rightarrow$ set of all executions $[P]$ under arbitrary behaviour of the database

• Semantics of $P$ when using $\mathcal{H}$:
  $[P, \mathcal{H}] = \{ X \in [P] \mid \text{history}(X) \in \mathcal{H} \}$

$P:
\begin{align*}
r1 &= \text{x.read}(); \\
r2 &= \text{x.read}(); \\
y.write(r1==r2);
\end{align*}$

$[P]:
\begin{align*}
x.\text{read}() &\rightarrow 42; & x.\text{read}() &\rightarrow 42; \\
x.\text{read}() &\rightarrow 42; & x.\text{read}() &\rightarrow 43; \\
y.\text{write}(1); & y.\text{write}(0);
\end{align*}$
Defining a consistency model

- **Operational specification**: by an idealised implementation
- **Axiomatic specification**: more declarative
Strong consistency operationally

- \( x: \sigma \)
- \( x: \sigma = 0 \)

- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentially in the receipt order
Strong consistency operationally

- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentially in the receipt order
Strong consistency operationally

- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentially in the receipt order
Strong consistency operationally

- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentially in the receipt order
Strong consistency operationally

- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentially in the receipt order
Strong consistency operationally

- Server with a single copy of all objects
- Clients send request to the server and wait for a reply
- Server processes operation sequentially in the receipt order
Strong consistency operationally

x.write(42)  
ok  
x: σ  
x: σ = 0 
x.write(42) 
x: σ = 42  
x.read : 42  
x.read  
42

Could write a formal operational semantics: maintain the state of the database, clients and sets of messages between them
Strong consistency operationally

- Consistency model = \{ H \mid \exists \text{ execution with history } H \text{ produced by the abstract implementation} \}

- **Sequential consistency**: one form of strong consistency

- Weaker than **linearizability**: takes into account the duration of operations
Operational specifications

• Let one understand intuitions behind implementations

• May become unwieldy for weaker consistency models

• Sometimes overspecify behaviour
Axiomatic specifications

- Choose a set of relations over events: \(r_1, \ldots, r_n\)
  Abstractly specify essential information about how operations are processed inside the system

- Abstract execution \((H, r_1, \ldots, r_n) = (E, so, r_1, \ldots, r_n)\)

- Choose a set of axioms \(\mathcal{A}\) constraining abstract executions

- Consistency model = \(\{H \mid \exists r_1, \ldots, r_n. (H, r_1, \ldots, r_n) \models \mathcal{A}\}\)
Axiomatic specifications

• Choose a set of relations over events: \( r_1, ..., r_n \)
  Abstractly specify essential information about how operations are processed inside the system

• Abstract execution \((H, r_1, ..., r_n) = (E, \text{so}, r_1, ..., r_n)\)

• Choose a set of axioms \( \mathcal{A} \) constraining abstract executions

• Consistency model = \( \{ H \mid \exists r_1, ..., r_n. (H, r_1, ..., r_n) \models \mathcal{A} \} \)

vs

Consistency model = \( \{ H \mid \exists \text{execution with history } H \text{ produced by the abstract implementation} \} \)
Axiomatic specifications

• Choose a set of relations over events: \( r_1, \ldots, r_n \)
  Abstractly specify essential information about how operations are processed inside the system

• Abstract execution \((H, r_1, \ldots, r_n) = (E, so, r_1, \ldots, r_n)\)

• Choose a set of axioms \( \mathcal{A} \) constraining abstract executions

• Consistency model = \( \{H \mid \exists r_1, \ldots, r_n. (H, r_1, \ldots, r_n) \models \mathcal{A}\} \)

\[\text{vs} \quad \text{Consistency model} = \{H \mid \exists \text{execution with history } H \text{ produced by the abstract implementation}\}\]
Axiomatic specifications

• Choose a set of relations over events: \( r_1, ..., r_n \)
  Abstractly specify essential information about how operations are processed inside the system

• Abstract execution \((H, r_1, ..., r_n) = (E, so, r_1, ..., r_n)\)

• Choose a set of axioms \( \mathcal{A} \) constraining abstract executions

• Consistency model = \( \{ H \mid \exists r_1, ..., r_n. (H, r_1, ..., r_n) \models \mathcal{A} \} \)

vs

Consistency model = \( \{ H \mid \exists \text{ execution with history } H \text{ produced by the abstract implementation} \} \)
Sequential consistency axiomatically

An SC history can be explained by a total order over all events: the order in which the server processes client operations.

- x.write(42)
- ok
- x.read
- 42
Sequential consistency axiomatically

An SC history can be explained by a total order over all events: the order in which the server processes client operations

Abstract execution: \((H, to) = (E, so, to)\), where \(to \subseteq E \times E\)

\[ SC = \{(E, so) \mid \exists \text{ total order } to. (E, so, to) \models \mathcal{A}_{SC}\} \]
(E, so, to) ⊨ A_{SC} \iff

1. so \subseteq to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to.
(E, so, to) ⊨ \mathcal{A}_{SC} \text{ iff}

1. so ⊆ to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

\forall e \in E. \text{type(obj(e))} = (\sigma_0, [\cdot]_{\text{val}}, [\cdot]_{\text{state}})
(E, so, to) ⊨ \mathcal{ASC} \text{ iff }

1. so \subseteq to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

\forall e \in E. \text{type(obj(e))} = (\sigma_0, \llbracket - \rrbracket_{\text{val}}, \llbracket - \rrbracket_{\text{state}})

\text{rval(e)} = \llbracket \text{op(e)} \rrbracket_{\text{val}}(\sigma)
(E, so, to) ⊨ ASC iff

1. so ⊆ to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

∀e ∈ E. \text{type}(\text{obj}(e)) = (σ_0, [\_]_{\text{val}}, [\_]_{\text{state}})

rval(e) = \text{[\text{op}(e)]}_{\text{val}}(σ)

σ = \text{[\text{op}(e_n)]}_{\text{state}}(...\text{[\text{op}(e_1)]}_{\text{state}}(σ_0))

e_1, ..., e_n = \text{to}^{-1}(e).\text{select(\text{obj}(e))}.\text{sort(to)}
\( (E, \text{so}, \text{to}) \models \mathcal{A}_\text{SC} \text{ iff } \)

1. \( \text{so} \subseteq \text{to} \)

2. The return value of each operation in \( E \) is computed from a state obtained by executing all operations on the same object preceding it in \( \text{to} \)

\[ \forall e \in E. \text{type}(\text{obj}(e)) = (\sigma_0, \llbracket - \rrbracket_{\text{val}}, \llbracket - \rrbracket_{\text{state}}) \]

\( \text{rval}(e) = \llbracket \text{op}(e) \rrbracket_{\text{val}}(\sigma) \)

\( \sigma = \llbracket \text{op}(e_n) \rrbracket_{\text{state}}(...\llbracket \text{op}(e_1) \rrbracket_{\text{state}}(\sigma_0)) \)

\( e_1, ..., e_n = \text{to}^{-1}(e).\text{select(\text{obj}(e))}.\text{sort(\text{to})} \)

Integer registers: a read returns the value written by the last preceding event in \( \text{to} \) (or 0 if there are none)

\[ x.\text{write}(0); \ x.\text{write}(42); \ x.\text{read}: 42 \]
(E, so, to) ⊨ A_{SC} \iff SC = \{(E, so) \mid \exists \text{to}. (E, so, to) \models A_{SC}\}

1. so ⊆ to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

∀e ∈ E. type(obj(e)) = (σ_0, [], [], [], [], [])

rval(e) = \[\text{op}(e)\]_{\text{val}}(σ)

σ = \[\text{op}(e_n)\]_{\text{state}}(...\[\text{op}(e_1)\]_{\text{state}}(σ_0))

e_1, ..., e_n = \text{to}^{-1}(e).\text{select}(\text{obj}(e)).\text{sort}(\text{to})

Integer registers: a read returns the value written by the last preceding event in to (or 0 if there are none)

x.write(0); x.write(42); x.read: 42
SC example

\[
SC = \{ (E, \text{so}) \mid \exists \text{to}. (E, \text{so}, \text{to}) \models \mathcal{A}_\text{SC} \}
\]

- x.read: 0
  - y.write(1)
    - z.write(2)
      - c.add(1)
- x.write(1)
  - c.add(1)
- x.write(1)
  - c.read: 1
  - z.read: 2
SC example

\[ SC = \{(E, so) \mid \exists to. (E, so, to) \models A_{SC}\} \]
• Got rid of messages between clients and the server, but didn't go far from the operational spec

• There's more difference for weaker models: complex processing can be concisely specified by axioms
Process A:

```python
x.write(1)
if (y.read() == 0)
    print "A wins"
```

Process B:

```python
y.write(1)
if (x.read() == 0)
    print "B wins"
```
Dekker example

Process A:
  x.write(1)
  if (y.read() == 0)
  print "A wins"

Process B:
  y.write(1)
  if (x.read() == 0)
  print "B wins"

Claim: under sequential consistency, there can be at most one winner
Assume there are two winners. Then there must exist an abstract execution for the history:

```
x.write(1)
if (y.read() == 0)
  print "A wins"
```

```
y.write(1)
if (x.read() == 0)
  print "B wins"
```

Need to construct a total order to
Assume there are two winners. Then there must exist an abstract execution for the history:

Process A:
- `x.write(1)`
- `if (y.read() == 0) print "A wins"

Process B:
- `y.write(1)`
- `if (x.read() == 0) print "B wins"

so $\subseteq$ to
Assume there are two winners. Then there must exist an abstract execution for the history:

**Process A:**
- `x.write(1)`
- `if (y.read() == 0)`
- `print "A wins"`

**Process B:**
- `y.write(1)`
- `if (x.read() == 0)`
- `print "B wins"`

```
x.write(1)     y.write(1)
   ↓            ↓
so, to        so, to
```
```
y.read(): 0     x.read(): 0
```
Assume there are two winners. Then there must exist an abstract execution for the history:

Process A:
```
x.write(1)
if (y.read() == 0)
    print "A wins"
```

Process B:
```
y.write(1)
if (x.read() == 0)
    print "B wins"
```
Assume there are two winners. Then there must exist an abstract execution for the history:

**Process A:**
- `x.write(1)`
- if `(y.read() == 0)`
  - print "A wins"

**Process B:**
- `y.write(1)`
- if `(x.read() == 0)`
  - print "B wins"

Reads return the most recent write in `to`, but this read doesn't see the write.
Assume there are two winners. Then there must exist an abstract execution for the history:

Process A:
```
x.write(1)
if (y.read() == 0)
    print "A wins"
```

Process B:
```
y.write(1)
if (x.read() == 0)
    print "B wins"
```

Reads return the most recent write in `to`, but this read doesn't see the write.
Assume there are two winners. Then there must exist an abstract execution for the history:

Process A:
```
x.write(1)
if (y.read() == 0)
    print "A wins"
```

Process B:
```
y.write(1)
if (x.read() == 0)
    print "B wins"
```

Reads return the most recent write in to, but this read doesn't see the write.
Assume there are two winners. Then there must exist an abstract execution for the history:

Process A:

```
x.write(1)
if (y.read() == 0)
  print "A wins"
```

Process B:

```
y.write(1)
if (x.read() == 0)
  print "B wins"
```

Reads return the most recent write in to, but this read doesn't see the write
Assume there are two winners. Then there must exist an abstract execution for the history:

Process A:
```
x.write(1)
if (y.read() == 0)
print "A wins"
```

Process B:
```
y.write(1)
if (x.read() == 0)
print "B wins"
```

But to must be acyclic, so no such total order exists - QED.
CAP theorem

No system with at least 2 processes can implement a read-write register with strong consistency, availability, and partition tolerance

- strong consistency = sequential consistency
- availability = all operations eventually complete
- partition tolerance = system continues to function under permanent network partitions
  (processes in different partitions can no longer communicate in any way)
CAP proof

No system with at least 2 processes can implement a read-write register with strong consistency, availability, and partition tolerance

- By contradiction: assume the desired system exists
- Run some experiments with the Dekker program
- Network is partitioned between the two processes

Process A:
- x.write(1)
- if (y.read() == 0)
- print "A wins"

Process B:
- y.write(1)
- if (x.read() == 0)
- print "B wins"
<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.write(1)</td>
<td></td>
</tr>
<tr>
<td>if (y.read() == 0)</td>
<td></td>
</tr>
<tr>
<td>print &quot;A wins&quot;</td>
<td></td>
</tr>
</tbody>
</table>

- Process A runs its code, process B is idle
• Process A runs its code, process B is idle

• Availability $\implies$ A must terminate and produce an execution $X_A$
Process A
x.write(1)
if (y.read() == 0)
    print "A wins"

Process B

- Process A runs its code, process B is idle
- Availability $\implies$ A must terminate and produce an execution $X_A$
- Sequential consistency $\implies$ $X_A$ must print "A wins"
Process A

```python
x.write(1)
if (y.read() == 0):
    print "A wins"
```

Process B

- Process B runs its code, process A is idle
- Availability $\implies$ B must terminate and produce an execution $X_B$
- Sequential consistency $\implies$ $X_B$ must print "B wins"

Process A

```python
y.write(1)
if (x.read() == 0):
    print "B wins"
```
Network is partitioned in both experiments: processes didn't receive any messages

$X_A; X_B$ is an execution of $A \parallel B$, i.e., Dekker

$X_A; X_B$ not SC $\Rightarrow$ contradiction, QED
Processes have to talk to each other (synchronise) to guarantee strong consistency.
Eventual consistency and replicated data types, operationally
System model

- Database system consisting of multiple replicas (= data centre, machine, mobile device)
- Each replica stores a copy of all objects
System model

- Replicas can communicate via channels

- **Asynchronous:** no bound on how quickly a message will be delivered
  (in particular, because of network partitions)

- **Reliable:** every message is eventually delivered
  (so every partition eventually heals)

- For now: replicas are reliable too
High availability

- Clients connect to a replica of their choice
High availability

- Clients connect to a replica of their choice

x.write(1)

y.write(1)
High availability

- Clients connect to a replica of their choice
- Replica has to respond to operations immediately, without communicating with others
High availability

- Clients connect to a replica of their choice
- Replica has to respond to operations immediately, without communicating with others
- Propagate effects to other replicas later

```plaintext
x.write(1)
ok
y.write(1)
ok
```
High availability

- Clients connect to a replica of their choice
- Replica has to respond to operations immediately, without communicating with others
- Propagate effects to other replicas later
- Always available, low latency, but may not be strongly consistent
High availability

- **Quiescent consistency**: if no new updates are made to the database, then replicas will eventually converge to the same state

- Later more precise and stronger formulations of eventual consistency
Replicated data types

• Need a new kind of replicated data type: object state now lives at multiple replicas

• Aka CRDTs: commutative, convergent, conflict-free
  Just one type: operation-based replicated data types

• Object ➔ Type ➔ Operation signature
  For now fix a single object and type
Sequential semantics recap

- Set of states \( \text{State} \)
- Initial state \( \sigma_0 \in \text{State} \)
- \( \llbracket \text{op} \rrbracket_{\text{val}} \in \text{State} \rightarrow \text{Value} \)
- \( \llbracket \text{op} \rrbracket_{\text{state}} \in \text{State} \rightarrow \text{State} \)
Replicated data types

Object state at a replica: $\sigma \in \text{State}$
Replicated data types

Object state at a replica: $\sigma \in \text{State}$

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

Object state at a replica: $\sigma \in \text{State}$

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

The operation affects a different state $\sigma'$!
Replicated data types

Object state at a replica: $\sigma \in \text{State}$

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

Object state at a replica: $\sigma \in \text{State}$

Return value: $\lfloor \text{op} \rfloor_{\text{val}} \in \text{State} \rightarrow \text{Value}$

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

Object state at a replica: \( \sigma \in \text{State} \)

Return value: \( [\text{op}]_{\text{val}} \in \text{State} \rightarrow \text{Value} \)

Effector: \( [\text{op}]_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State}) \)
Replicated data types

Object state at a replica: $\sigma \in \text{State}$

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

Effector: $[\text{op}]_\text{eff} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
$\sigma \ 
\boxed{\text{op}} \ 
\boxed{\text{val}} \ 
\llbracket \text{op} \rrbracket_{\text{val}}(\sigma) = \sigma \ 
\llbracket \text{read() \rrbracket_{\text{val}}(\sigma) = \sigma \ 
\llbracket \text{read() \rrbracket_{\text{eff}}(\sigma) = \lambda\sigma.\sigma \ 
\text{State} = \mathbb{N} \ 
\llbracket \text{op} \rrbracket_{\text{eff}}(\sigma)(\sigma') = \lambda\sigma. \sigma'$
\[ \langle \text{add(100)} \rangle_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100) \]
\[ \text{Counter} \]

\[ \sigma \]

\[ \text{op} \]

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

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

\[ 50 \]

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

\[ \left[ \text{add}(100) \right]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100) \]
\[
\langle \text{add}(100) \rangle_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100)
\]
Counter

\[ \boxed{\text{add(100)}} \text{ eff}(\sigma) = \lambda\sigma'. (\sigma + 100) \]
count = 0

add(100)

count = 0

add(200)
count = 0

add(100)

\( \lambda \sigma'. 100 \)

count = 100

count = 0

add(200)

\( \lambda \sigma'. 200 \)

count = 200
Quiescent consistency violated: all updates have been delivered, yet replicas will never converge.
Ensuring quiescent consistency

- **Effectors have to commute:**

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

- **Convergence:** replicas that received the same sets of updates end up in the same state
  (even when messages are received in different orders)
Ensuring quiescent consistency

- **Effectors have to commute:**

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

- **Convergence:** replicas that received the same sets of updates end up in the same state
  (even when messages are received in different orders)

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

- **Quiescent consistency:** if no new updates are made to the database, then replicas will eventually converge to the same state
  (because update get eventually delivered)
Replicated data types

- Counter
- Last-writer-wins register
- Multi-valued register
- Add-wins set
- Remove-wins set
- List
- ...

...
Read-write register

write(1)

write(2)
Read-write register

write(1)  

write(2)
Read-write register

write(1)  Conflict!  write(2)
Read-write register

- No right or wrong solutions: depends on the application requirements
- E.g., could report the conflict to the user: multi-valued register
Last-writer-wins register

- Shared memory: an arbitrary write will win
- Conflict arbitrated using timestamps: last write wins
- Link to shared-memory consistency models
Last-writer-wins register

State $= \text{Value} \times \text{Timestamp}$

$\llbracket \text{read()} \rrbracket_{\text{val}}(v, t) = v$
Last-writer-wins register

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

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

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

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

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

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

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

\[
\begin{align*}
\text{let } t_{\text{new}} &= \text{newUniqueTS}() \text{ in } \\
\lambda(v', t'). & \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)
\end{align*}
\]
Last-writer-wins register

\[
\begin{align*}
\text{let } t_{\text{new}} &= \text{newUniqueTS}() \text{ in} \\
\lambda(v', t'). \text{if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)
\end{align*}
\]
Effectors are commutative: the write with the highest timestamp wins regardless of the order of application.
Generating timestamps

• Can use wall-clock time at the machine

• But can lead to strange results when clocks are out of sync
write(1)
write(1) → read: 1 → write(2)
write(1)

read: 1

write(2)

$\text{t}_1 > \text{t}_2$
write(1) → t₁
read: 1
write(2) → t₂

t₁ > t₂
write(1)

$\text{t}_1$

$\text{t}_1 > \text{t}_2$

write(2)

read: 1

$\text{t}_2$

read: 1
- Undesirable: 2 was meant to supersede 1
- Undesirable: 2 was meant to supersede 1
- Use logical (Lamport) clocks instead
Lamport clock

Replica maintains a counter, incremented on each operation:

time = 1
Lamport clock

Replica maintains a counter, incremented on each operation:

- time = 1
- write(1) 1
- time = 2
Lamport clock

Replica maintains a counter, incremented on each operation:

\[
\begin{align*}
\text{time} &= 1 \\
\text{write}(1) &= 1 \\
\text{time} &= 2 \\
\text{write}(2) &= 2
\end{align*}
\]
Lamport clock

Replica maintains a counter, incremented on each operation:

- time = 1
- write(1) 1
- time = 2
- write(2) 2
- time = 1
- write(1) 1
Lamport clock

Replica maintains a counter, incremented on each operation:

Timestamps need to be unique: \( ts = (\text{CounterValue}, \text{ReplicaID}) \)
Lamport clock

Replica maintains a counter, incremented on each operation:

Timestamps need to be unique: \( ts = (\text{CounterValue}, \text{ReplicaID}) \)
Lamport clock

Replica maintains a counter, incremented on each operation:

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Operation</th>
<th>Replica ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>write(1)</td>
<td>r₁</td>
</tr>
<tr>
<td>2</td>
<td>write(2)</td>
<td>r₁</td>
</tr>
<tr>
<td>1</td>
<td>write(1)</td>
<td>r₂</td>
</tr>
</tbody>
</table>
```

Timestamps need to be unique: \( ts = (\text{CounterValue}, \text{ReplicaID}) \)

\[(c₁, r₁) < (c₂, r₂) \iff c₁ < c₂ \lor (c₁ = c₂ \land r₁ < r₂)\]
Lamport clock

Replica maintains a counter, incremented on each operation:

Timestamps need to be unique: \( ts = (\text{CounterValue}, \text{ReplicaID}) \)

\[(c_1, r_1) < (c_2, r_2) \iff c_1 < c_2 \lor (c_1 = c_2 \land r_1 < r_2)\]
time = t_1

write(1)
write(1)
(time = t₁)
(time = t₁ + 1)
write(1)

(time = t_1)

(t_1, r_1)

(time = t_2)

(time = t_1 + 1)
When receiving an effector, bump up your clock above its timestamp.
When receiving an effector, bump up your clock above its timestamp.
When receiving an effector, bump up your clock above its timestamp.
When receiving an effector, bump up your clock above its timestamp.
Replicated set

cart = \{book\}

cart.add(\textit{book})

cart.remove(\textit{book})
Replicated set

cart = \{book\}

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

cart = \{ book \}

cart.add(book)          Conflict!          cart.remove(book)

Should the remove cancel the concurrent add?
Depends on application requirements
Replicated set

cart = \{book\}

Last writer wins: choose based on operation time-stamps

Remove wins: cart = \emptyset

Add wins: cart = \{book\}
Add-wins set

cart = \{\text{book}\}

cart.add(\text{book})

cart.remove(\text{book})

cart = \{\text{book}\}
Add-wins set

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

- `cart = {book}`

- `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 \textit{add()} creates a new element instance:

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

$$\langle \text{add}(v) \rangle_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' \cup \{(v, \text{uniqueid}())\})$$
\{(\textit{book}, 1)\}

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

Instance ids ignored when reading the set:

\[
\mathcal{V}_{val}(\sigma) = \{v \mid \exists \text{id.} \ (v, \text{id}) \in \sigma\}
\]
add(\text{book})

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

\{ (\text{book}, 1), (\text{book}, 2) \}

remove(\text{book})

\{ (\text{book}, 1) \}
remove(v) removes all currently present instances of x:

\[
\left[\text{remove}(v)\right]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' \setminus \{(v, \text{id}) \in \sigma\})
\]
remove(v) removes all currently present instances of x:

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

\[
\llbracket \text{remove}(v) \rrbracket_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' \setminus \{(v, \text{id}) \in \sigma\})
\]
\{ (book, 1) \} \\
\text{add(} book \text{)} \\
\{ (book, 1), (book, 2) \} \\
\{ (book, 2) \}

\{ (book, 1) \} \\
\text{remove(} book \text{)} \\
\emptyset
add(book) \{ (book,1) \}

\{ (book,1), (book,2) \}

\{ (book,2) \}

\{ (book,2) \}

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

Effectors commutative \rightarrow replicas converge
Take-aways

- Need to ensure commutativity to guarantee quiescent consistency

- Need to make choices about how to resolve conflicts
Replicated data type uses

- Provided by some data stores:
  - redis
  - riak

- Implemented by programmers on their own:
  - PayPal
  - TomTom
  - SoundCloud
Reasoning about data consistency in distributed systems

Alexey Gotsman

IMDEA Software Institute, Madrid, Spain

Collaborative editing: at the core - list data type (of formatted characters)
Operational specification

- Given a database with a set of objects of replicated data types

- Eventual consistency model = set of all histories produced by arbitrary client interactions with the data type implementations (with any allowed message deliveries)

- Implies quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state
Eventual consistency and replicated data types, axiomatically
Anomalies

c.add(1)

... 

c.read(): ?
Anomalies

c.add(1)

...}

c.read(): 0
Can be disallowed if the client sticks to the same replica:
Read Your Writes guarantee
Anomalies

access.write(all)

access.write(noboss)

post.write(photo)
Anomalies

access.write(all)

access.write(noboss)

post.write(photo)
Anomalies

access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all
Anomalies

access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() :

access.read() : all
Anomalies

Causality violation: disallowed by causal consistency
Anomalies

Causality violation: disallowed by causal consistency
Specification

- Lots of replicated data type implementations: e.g., can send snapshots of object states instead of operations
- Lots of message delivery guarantees: different implementations of causal consistency
- Want specifications that abstract from implementation details: both replicated data types and anomalies
Axiomatic specifications

• Choose a set of relations over events: $r_1, \ldots, r_n$
  Abstractly specify essential information about how operations are processed inside the system

• Abstract execution $(H, r_1, \ldots, r_n)$

• Choose a set of axioms $\mathcal{A}$ constraining abstract executions

• Consistency model = \{ $H$ | $\exists$ $r_1, \ldots, r_n$. $(H, r_1, \ldots, r_n) \models \mathcal{A}$ \}
(E, so) | ∃ total order to. (E, so, to) satisfies:

1. so ⊆ to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to
Sequential consistency

\[(E, so) \mid \exists \text{ total order } to. (E, so, to) \text{ satisfies:}\]

1. \(so \subseteq to\)

2. The return value of each operation in \(E\) is computed from a state obtained by executing all operations on the same object preceding it in to
1. so $\subseteq$ to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to
Sequential consistency

1. \( \subseteq \) to

2. The return value of each operation in \( E \) is computed from a state obtained by executing all operations on the same object preceding it in to

Return value axiom: replicated data types
Execution: (E, so, vis, ar)

access.write(all)

access.write(noboss) → post.read() : photo

post.write(photo)

access.read() : all
Execution: (E, so, vis, ar)

access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all
Execution: (E, so, vis, ar)

access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all

The order of requests by the same session
Declaratively specify ways in which the database processes requests.
access.write(noboss)

post.write(photo)

access.write(all)

post.read() : photo

access.read() : all
access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all

Delivered?

so

so

so
access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

Visible?

Delivered?

access.read() : all
Execution: (E, so, vis, ar)

access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all

Visibility relation
Execution: (E, so, \textit{vis}, ar)

- `access.write(all)`
- `access.write(noboss)`
- `post.write(photo)`
- `post.read() : photo`
- `access.read() : all`

\textit{vis} is irreflexive and acyclic
Execution: $(E, \text{so, vis, ar})$

- `access.write(all)`
- `access.write(noboss)`
- `post.write(photo)`
- `post.read() : photo`
- `access.read() : noboss`

**Visibility relation**

- `vis` is irreflexive and acyclic
System includes a time-stamping mechanism that can be used in conflict resolution.
System includes a time-stamping mechanism that can be used in conflict resolution.
System includes a time-stamping mechanism that can be used in conflict resolution
Execution: \((E, \text{ so}, \text{ vis}, \text{ ar})\)

System includes a time-stamping mechanism that can be used in conflict resolution.

\(\text{ar} \text{ is total on } E \text{ and } \text{vis} \subseteq \text{ar}\)
Data type specification

- How do I compute the return value of an event $e$?
- Only actions on the same object visible to $e$ are important: have been delivered to the replica performing $e$

\[
\text{access.write(all)}
\]

\[
\text{access.write(noboss)}
\]

\[
\text{post.write(photo)}
\]

\[
\text{post.read() : photo}
\]

\[
\text{access.read() : noboss}
\]
Data type specification

- How do I compute the return value of an event $e$?
- Only actions on the same object visible to $e$ are important: have been delivered to the replica performing $e$
Data type specification

\[ \forall e \in E. \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context(e)}) \]

\[ \text{access.write(all)} \]

\[ \text{access.write(noboss)} \]

\[ \text{access.read() : noboss} \]
Data type specification

\[ F : \text{context of } e \rightarrow \text{return value of } e \]

\[ \forall e \in E. \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context}(e)) \]

access.write(all) [F for Last-Writer-Wins registers: sort all actions according to \( ar \) and return the last value written]

access.write(noboss)

access.read() : noboss
Data type specification

F: context of e → return value of e

∀e ∈ E. rval(e) = F_{type(obj(e))}(context(e))

access.write(all)
What gets taken into account depends only on vis

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all
Counter

F: context of $e$ → return value of $e$

c.add(1)  
c.add(2)  
c.add(3)

vis  vis  vis

c.read(): 6

F: reads return the sum of all additions in the context
Counter

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

Relations between events in the context don't matter
Counter with decrements

\[ \text{F: context of } e \rightarrow \text{return value of } e \]

c.add(1) \quad c.add(2) \quad c.subtract(4)

\[ \text{c.read(): -1} \]

\[ \text{F: reads return additions minus subtractions} \]
Multi-valued register

\[ F: \text{context of } e \rightarrow \text{return value of } e \]

\[ x.\text{write}(1) \quad x.\text{write}(2) \quad x.\text{write}(3) \]

\[ x.\text{read}(): ? \]

\[ F: \text{reads return the set of all conflicting writes} \]
Multi-valued register

\[ F: \text{context of } e \rightarrow \text{return value of } e \]

\[ \begin{align*}
\text{x.write}(1) & \quad \text{x.write}(2) & \quad \text{x.write}(3) \\
\text{vis} & \quad \text{vis} & \quad \text{vis} \\
\text{x.read}(): ? &
\end{align*} \]

\[ F: \text{reads return the set of all conflicting writes} \]
Multi-valued register

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

F: reads return the set of all conflicting writes
Multi-valued register

F: context of e → return value of e

x.write(1) \rightarrow x.write(2) \rightarrow x.write(3)

x.read(): ?

F: reads return the set of all conflicting writes
Multi-valued register

\[ F: \text{context of } e \rightarrow \text{return value of } e \]

\[ \begin{align*}
    \text{x.write}(1) & \quad \text{x.write}(2) \quad \text{x.write}(3) \\
    \text{x.read}(): \{1, 3\} & \\
\end{align*} \]
Multi-valued register

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

x.write(1) $\xrightarrow{\text{vis}}$ x.write(2) $\xrightarrow{\text{vis}}$ x.write(3) $\xrightarrow{\text{vis}}$ x.read(): \{1, 3\}

F: discard all writes seen by a write
Multi-valued register

F: context of e → return value of e

x.write(1) → x.write(2) → x.write(3)

x.read(): {3}

F: discard all writes seen by a write
Add-wins set

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

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

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

\[ F : \text{context of } e \rightarrow \text{return value of } e \]

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

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

set.add(\textit{book})

\textit{set.add(\textit{book})}

set.remove(\textit{book})

\textit{set.remove(\textit{book})}

\textit{set.read()} : ?
Add-wins set

\[ \text{F: context of } e \rightarrow \text{return value of } e \]

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

F: context of e → return value of e

Add-wins set

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

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

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

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

F: context of e → return value of e


vis  vis  vis

set.read() : ∅

F: cancel all adds seen by a remove
Data type specification

\[ F: \text{context of } e \rightarrow \text{return value of } e \]

\[ \forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e)) \]
"No causal cycles" axiom

- \textit{so} \cup \textit{vis} is acyclic: no causal cycles/out-of-thin-air values
- \textit{so} and \textit{vis} consistent with execution order
"No causal cycles" axiom

- \( \text{so} \cup \text{vis} \text{ is acyclic: no causal cycles/out-of-thin-air values} \)
- \( \text{so and vis} \text{ consistent with execution order} \)
- Could result from speculative execution, uncommon in distributed systems
"No causal cycles" axiom

\[
x.\text{read}(): 42 \quad \text{vis} \quad y.\text{read}(): 42
\]

\[
y.\text{write}(42) \quad \text{vis} \quad x.\text{write}(42)
\]

- \( \text{so} \cup \text{vis} \) is acyclic: no causal cycles/out-of-thin-air values
- \text{so} and \text{vis} consistent with execution order
- Could result from speculative execution, uncommon in distributed systems
- Some forms allowed by shared-memory models (ARM, C++, Java): defining semantics is an open problem
Eventual visibility

∀e ∈ E. e \xrightarrow{vis} f for all but finitely many f ∈ E
∀e ∈ E. e \xrightarrow{vis} f for all but finitely many f ∈ E
Eventual consistency summary

The set of histories \((E, \text{so})\) such that for some \(\text{vis, ar}\):

- Return values consistent with data type specs:
  \[
  \forall e \in E. \text{rval}(e) = F_{\text{type}(	ext{obj}(e))}(\text{context}(e))
  \]

- No causal cycles: \(\text{so} \cup \text{vis}\) is acyclic

- Eventual visibility:
  \[
  \forall e \in E. e \xrightarrow{\text{vis}} f \text{ for all but finitely many } f \in E
  \]
Eventual consistency summary

The set of histories \((E, so)\) such that for some \(vis, ar\):

- Return values consistent with data type specs:
  \[
  \forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e))
  \]

- No causal cycles: \(so \cup vis\) is acyclic

- Eventual visibility:
  \[
  \forall e \in E. e \xrightarrow{vis} f \text{ for all but finitely many } f \in E
  \]

**Quiescent consistency**: if no new updates are made to the database, then replicas will eventually converge to the same state
**Quiescent consistency:** if no new updates are made to the database, then replicas will eventually converge to the same state.
Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- **Convergence**: events with the same context return the same value: \( \forall e \in E. \, \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context(e)}) \)
Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state.

- **Convergence**: events with the same context return the same value: \( \forall e \in E. \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context(e)}) \)

- **Assumption**: deleting read-only operations from the context doesn't change the return value: \( F(X) = F(X-\text{ReadOnlyEvents}) \)
**Quiescent consistency:** if no new updates are made to the database, then replicas will eventually converge to the same state

- **Convergence:** events with the same context return the same value: \( \forall e \in E. \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context(e)}) \)

- **Assumption:** deleting read-only operations from the context doesn't change the return value: \( F(X) = F(X-\text{ReadOnlyEvents}) \)

```
vis
- c.add(1)
  - vis
    - c.add(2)
      - vis
      - c.read(): 3
      - c.add(1)
        - vis
        - c.add(2)
          - vis
          - c.read(): 3
```
**Quiescent consistency:** if no new updates are made to the database, then replicas will eventually converge to the same state

- **Convergence:** events with the same context return the same value: \( \forall e \in E. \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context(e)}) \)

- **Assumption:** deleting read-only operations from the context doesn't change the return value: \( F(X) = F(X-\text{ReadOnlyEvents}) \)
Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state.

- **Convergence:** events with the same context return the same value: \( \forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e)) \)

- **Assumption:** deleting read-only operations from the context doesn't change the return value: \( F(X) = F(X\text{-ReadOnlyEvents}) \)

- **Convergence':** two operations with the same context projection to updates return the same value
Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- **Convergence**: events with the same context return the same value: \( \forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e)) \)

- **Assumption**: deleting read-only operations from the context doesn't change the return value: \( F(X) = F(X-\text{ReadOnlyEvents}) \)

- **Convergence'**: two operations with the same context projection to updates return the same value

- **Eventual visibility**: each update is seen by all but finitely many ops
Quiescent consistency: if no new updates are made to the database, then replicas will eventually converge to the same state

- **Convergence**: events with the same context return the same value: $\forall e \in E. \text{rval}(e) = F_{\text{type}(|\text{obj}(e)|)}(\text{context}(e))$

- **Assumption**: deleting read-only operations from the context doesn't change the return value: $F(X) = F(X-\text{ReadOnlyEvents})$

- **Convergence'**: two operations with the same context projection to updates return the same value

- **Eventual visibility**: each update is seen by all but finitely many ops

- **Assuming finitely many updates**, all but finitely many ops will see all of these updates
**Quiescent consistency**: if no new updates are made to the database, then replicas will eventually converge to the same state.

- **Convergence**: events with the same context return the same value: \(\forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e))\)

- **Assumption**: deleting read-only operations from the context doesn't change the return value: \(F(X) = F(X-\text{ReadOnlyEvents})\)

- **Convergence'**: two operations with the same context projection to updates return the same value

- **Eventual visibility**: each update is seen by all but finitely many ops

- **Assuming finitely many updates, all but finitely many ops will see all of these updates**

- **Quiescent consistency**: assuming finitely many updates, all but finitely many operations on a given object return values computed based on the same context: same op \(\implies\) same rval
Eventual consistency summary

The set of histories \((E, \text{so})\) such that for some \(\text{vis}, \text{ar}\):

- Return values consistent with data type specs:
  \(\forall e \in E. \text{rval}(e) = \text{F}_{\text{type(obj(e))}}(\text{context(e)})\)

- No causal cycles: \(\text{so} \cup \text{vis}\) is acyclic

- Eventual visibility:
  \(\forall e \in E. e \xrightarrow{\text{vis}} f\) for all but finitely many \(f \in E\)
Eventual consistency summary

The set of histories \((E, so)\) such that for some \(vis, ar\):

- Return values consistent with data type specs:
  \[\forall e \in E. \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context(e)})\]

- No causal cycles: \(so \cup vis\) is acyclic

- Eventual visibility:
  \[\forall e \in E. e \xrightarrow{vis} f\] for all but finitely many \(f \in E\)

Stronger than quiescent consistency, but still weak

Strengthen consistency by adding additional axioms on \(vis\) and \(ar\)
Why is this spec sound wrt implementations?

The set of histories \((E, so)\) such that for some \(vis, ar\):

- Return values consistent with data type specs:
  \[
  \forall e \in E. \ rval(e) = F_{\text{type(obj(e))}}(\text{context(e)})
  \]

- No causal cycles: \(so \cup vis\) is acyclic

- Eventual visibility:
  \[
  \forall e \in E. \ e \xrightarrow{vis} f \text{ for all but finitely many } f \in E
  \]

Stronger than quiescent consistency, but still weak

Strengthen consistency by adding additional axioms on \(vis\) and \(ar\)
Specification soundness

The set of all histories \((E, so)\) such that for some vis, ar
the abstract execution \((E, so, vis, ar)\) satisfies
consistency axioms \(\mathcal{A}\)
Specification soundness

The set of all histories \((E, so)\) such that for some \(vis, ar\) the abstract execution \((E, so, vis, ar)\) satisfies consistency axioms \(\mathcal{A}\)

The set of all histories \((E, so)\) produced by arbitrary client interactions with the data type implementations with any allowed message deliveries
Specification soundness

The set of all histories \((E, so)\) such that for some \(\text{vis}, \text{ar}\) the abstract execution \((E, so, \text{vis}, \text{ar})\) satisfies consistency axioms \(\mathcal{A}\)

\[\supseteq\]

The set of all histories \((E, so)\) produced by arbitrary client interactions with the data type implementations with any allowed message deliveries
Specification soundness

The set of all histories \((E, so)\) such that for some \(\text{vis}, \text{ar}\) the abstract execution \((E, so, \text{vis}, \text{ar})\) satisfies consistency axioms \(\mathcal{A}\)

\(\supseteq\)

The set of all histories \((E, so)\) produced by arbitrary client interactions with the data type implementations with any allowed message deliveries

- \(\forall\) concrete execution of the implementation with a history \((E, so)\)
- \(\exists\ \text{vis, ar}. \ (E, so, \text{vis}, \text{ar})\) satisfies the axioms \(\mathcal{A}\)
Specification soundness

- Proofs depend on replicated data types
- Example: replicated counters and last-writer-wins registers
- There are also generic proof techniques that work for whole classes of data types

∀ concrete execution of the implementation with a history \((E, so)\)

∃ vis, ar. \((E, so, vis, ar)\) satisfies the axioms \(A\)
Specification soundness

- Proofs depend on replicated data types
- Example: replicated counters and last-writer-wins registers
- There are also generic proof techniques that work for whole classes of data types

∀ concrete execution of the implementation with a history \((E, so)\)

∃ vis, ar. \((E, so, vis, ar)\) satisfies the axioms \(\mathcal{A}\)
Constructing vis

e \xrightarrow{\text{vis}} f \iff \text{effector of } e \text{ delivered to replica of } f \text{ before } f \text{ is executed}
Constructing vis

e $\xrightarrow{\text{vis}}$ f $\iff$ effector of e delivered to replica of f before f is executed
Constructing $\text{vis}$

$e \xrightarrow{\text{vis}} f \iff$ effector of $e$ delivered to replica of $f$ before $f$ is executed
so $\cup$ vis is acyclic?

\[ e \xrightarrow{\text{vis}} f \iff \text{effector of } e \text{ delivered to replica of } f \text{ before } f \text{ is executed} \]
so $\cup \text{vis}$ is acyclic?

e $\xrightarrow{\text{vis}}$ f $\lor$ e $\xrightarrow{\text{so}}$ f $\implies$ e was issued before f in the operational execution
∀e ∈ E. e \xrightarrow{\text{vis}} f for all but finitely many f ∈ E
∀e ∈ E. e ↦^vis f for all but finitely many f ∈ E
∀e ∈ E. e \xrightarrow{\text{vis}} f \text{ for all but finitely many } f \in E
\forall e \in E. e \xrightarrow{\text{vis}} f \text{ for all but finitely many } f \in E

- Channels are reliable (every partition eventually heals) \iff the effector of e is eventually delivered to \( r_2 \)
\( \forall e \in E. e \xrightarrow{\text{vis}} f\) for all but finitely many \(f \in E\)

- Channels are reliable (every partition eventually heals) \(\Rightarrow\) the effector of \(e\) is eventually delivered to \(r_2\)
- From some point on, all events \(f_i\) at the replica \(r_2\) see \(e\)
\( \forall e \in E. e \rightarrow^\text{vis} f \) for all but finitely many \( f \in E \)

- Channels are reliable (every partition eventually heals) \( \implies \) the effector of \( e \) is eventually delivered to \( r_2 \)
- From some point on, all events \( f_i \) at the replica \( r_2 \) see \( e \)
- True for any replica \( \implies \) only finitely many events don't see \( e \)
Correctness of counters

∀e ∈ E. rval(e) = F_{\text{type(obj(e))}}(\text{context(e)})

c.add(1)       c.add(2)       c.add(3)

vis           vis           vis

\text{c.read()}: 6

\text{F: reads return the sum of all additions in the context}
Correctness of counters

c.read: ?
Correctness of counters

A read returns the value of the counter at the replica:

\[
\left[\text{read()}\right]_{\text{val}}(\sigma) = \sigma
\]
Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it.
Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it

\[
\llbracket \text{add}(v) \rrbracket_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + v)
\]
Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it

\[ \llbracket \text{add}(v) \rrbracket_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + v) \]
Invariant: the value of a counter at a replica is the sum of all increments of the counter delivered to it

= increments visible to the read, QED.
Every event e gets assigned a timestamp $t_e$ from a logical Lamport clock

$$e \xrightarrow{ar} f \iff t_e < t_f$$
Every event $e$ gets assigned a timestamp $t_e$ from a logical Lamport clock.

$$e \xrightarrow{ar} f \iff t_e < t_f$$
vis \subseteq ar

Every event e gets assigned a timestamp \( t_e \) from a logical Lamport clock

\[ e \xrightarrow{ar} f \iff t_e < t_f \]
Every event $e$ gets assigned a timestamp $t_e$ from a logical Lamport clock

$e \xrightarrow{ar} f \iff t_e < t_f$
Every event e gets assigned a timestamp $t_e$ from a logical Lamport clock

e $\xrightarrow{ar} f \iff t_e < t_f$
Every event $e$ gets assigned a timestamp $t_e$ from a logical Lamport clock

$$e \xrightarrow{\text{ar}} f \iff t_e < t_f$$
Correctness of registers

\[ \forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e)) \]

\[ \text{x.write}(1) \xrightarrow{\text{ar}} \text{x.write}(2) \xrightarrow{\text{vis}} \text{x.read}() : 2 \xrightarrow{\text{vis}} \text{x.write}(2) \]

F: reads return the last value in ar
Correctness of registers

\[ x.\text{read}: ? \]
Correctness of registers

A read returns the value part of the register at the replica:

$$\llbracket \text{read()} \rrbracket_{\text{val}}(v, t) = v$$
Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes.
Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

\[
\llbracket \text{write}(v_{\text{new}}) \rrbracket_{\text{eff}}(v, t) = \text{let } (t_{\text{new}} = \text{newUniqueTS}()) \text{ in } \\
\lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)
\]
Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes.

$$\llbracket \text{write}(v_{\text{new}}) \rrbracket_{\text{eff}}(v, t) = \text{let } (t_{\text{new}} = \text{newUniqueTS()} ) \text{ in } \lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)$$
Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

\[ \text{\texttt{\([\text{write}(v_{\text{new}})]_{\text{eff}}(v, t) = \text{let } (t_{\text{new}} = \text{newUniqueTS}()) \text{ in } \lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)\) } } \]
Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes

$$\llbracket \text{write}(v_{\text{new}}) \rrbracket_{\text{eff}}(v, t) = \text{let } (t_{\text{new}} = \text{newUniqueTS}()) \text{ in } \lambda(v', t'). \text{ if } t_{\text{new}} > t' \text{ then } (v_{\text{new}}, t_{\text{new}}) \text{ else } (v, t)$$
Correctness of registers

Invariant: the value of a register at a replica is the one with the highest timestamp out of all delivered writes = the last write in arbitration out of the ones visible to the read, QED.
Proof technique summary

- ∀ concrete execution of the implementation with a history (E, so)
- ∃ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathcal{A}$
Proof technique summary

- ∀ concrete execution of the implementation with a history (E, so)
- ∃ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathcal{A}$

- Construct vis from message deliveries and ar from timestamps
Proof technique summary

- ∀ concrete execution of the implementation with a history \((E, so)\)
- ∃ vis, ar. \((E, so, vis, ar)\) satisfies the axioms \(\mathcal{A}\)

- Construct vis from message deliveries and ar from timestamps
- Prove invariants relating replica state with message deliveries: the value of a counter at a replica is the sum of all increments of the counter delivered to it
Proof technique summary

• ∀ concrete execution of the implementation with a history (E, so)
• ∃ vis, ar. (E, so, vis, ar) satisfies the axioms $\mathcal{A}$

• Construct vis from message deliveries and ar from timestamps

• Prove invariants relating replica state with message deliveries: the value of a counter at a replica is the sum of all increments of the counter delivered to it

• Use the invariants to prove that return values of operations correspond to data type specs
In-between eventual and strong consistency
Eventual consistency summary

The set of histories \((E, \text{so})\) such that for some \(\text{vis}\), ar:

- Return values consistent with data type specs:
  \[\forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e))\]

- No causal cycles: \(\text{so} \cup \text{vis}\) is acyclic

- Eventual visibility:
  \[\forall e \in E. e \xrightarrow{\text{vis}} f\] for all but finitely many \(f \in E\)
Eventual consistency summary

The set of histories \((E, so)\) such that for some \(vis, ar\):

- Return values consistent with data type specs:
  \[
  \forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e))
  \]

- No causal cycles: \(so \cup vis\) is acyclic

- Eventual visibility:
  \[
  \forall e \in E. e \xrightarrow{vis} f \text{ for all but finitely many } f \in E
  \]

Stronger than quiescent consistency, but still weak

Strengthen consistency by adding additional axioms on \(vis\) and \(ar\)
Consistency zoo

Eventual consistency
Session guarantees
Causal consistency
Prefix consistency
Sequential consistency
Consistency zoo

- Eventual consistency
- Session guarantees
- Causal consistency
- Prefix consistency
- Sequential consistency

Keep soundness justifications informal: can be shown using previous techniques
c.add(100)

so

c.read(): 0
Read Your Writes

c.add(100)

so

c.read(): 0
c.add(100)

so

c.read(): 0
Read Your Writes

```python
c.add(100)
c.read(): 100
```

- An operation sees all prior operations by the same process
- **Session guarantees:** clients only accumulate information
Read Your Writes

• An operation sees all prior operations by the same process

• **Session guarantees:** clients only accumulate information

• Implementation: client sticks to the same replica
Monotonic Reads

c.add(100)

so

vis

c.add(100)

vis

c.read(): 100
Monotonic Reads

- An operation sees what prior operations by the same session see
Monotonic Reads

- An operation sees what prior operations by the same session see
- Implementation: client sticks to the same replica
Causal consistency

access.write(all)

access.write(noboss)  post.read() : photo

post.write(photo)  access.read() : all

Disallows causality violation anomaly
access.write(noboss)

post.write(photo)

access.read() : all

post.read() : photo

access.write(all)

so

unintuitive: chain of so and vis edges from write(noboss) to the read: write happened before the read

mandate that all actions that happened before an action be visible to it
Causal consistency

access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all

Unintuitive: chain of so and vis edges from write(noboss) to the read: write happened before the read

Mandate that all actions that happened before an action be visible to it
Causal consistency

Unintuitive: chain of so and vis edges from write(noboss) to the read: write happened before the read

Mandate that all actions that happened before an action be visible to it
Causal consistency

access.write(all)

access.write(noboss)

post.write(photo)

post.read() : photo

access.read() : all

\((so \cup vis)^+ \subseteq vis\)

Implies session guarantees: so \(\subseteq vis\) and vis; so \(\subseteq vis\)
Clients stick to the same replica
Clients stick to the same replica
Clients stick to the same replica
Cannot deliver an operation before delivering its causal dependencies
Replica order \texttt{ro}: the order in which operations are issued at a replica.
Delivery order `del`: one operation got delivered before another was issued.
• Causal dependencies of e: $hb^{-1}(e)$

• An op can only be delivered after all its causal dependencies

• Implementations summarise dependencies concisely

$hb = (ro \cup del)^+$
Dekker example

\begin{align*}
x.\text{write}(1) & \quad \text{so} \quad y.\text{read}(): 0 \\
\text{vis} & \quad \text{so} \quad \text{vis} \\
y.\text{write}(1) & \quad \text{so} \quad x.\text{read}(): 0 \\
\end{align*}
Dekker example

Implementations: updates delivered later
Independent reads of independent writes (IRIW)
Independent reads of independent writes (IRIW)

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

\[
\text{vis} \quad \text{vis} \quad \text{so} \quad \text{so}
\]

\[
x.\text{write}(1) \quad y.\text{read}: 0 \quad x.\text{read}: 0
\]

\[
\text{x written before y} \quad \text{y written before x}
\]
Independent reads of independent writes (IRIW)

Implementations: no causal dependency between the two writes

→ can be delivered in different orders at different replicas
Independent reads of independent writes (IRIW)

\( x.\text{write}(1) \rightarrow y.\text{write}(1) \)

\( x.\text{read}: 1 \)

\( y.\text{read}: 1 \)

\( x \text{ written before y} \)

\( y \text{ written before x} \)
Independent reads of independent writes (IRIW)

\[
x.\text{write}(1) \rightarrow y.\text{write}(1) \\
x.\text{read}: 1 \\
y.\text{read}: 0 \\
x \text{ written before } y
\]

\[
x.\text{read}: 0 \\
y.\text{read}: 1 \\
y \text{ written before } x
\]

Not sequentially consistent
Independent reads of independent writes (IRIW)

x.write(1) → y.write(1)

x.read: 1
y.read: 0

x written before y

y.read: 1
x.read: 0

y written before x

Not sequentially consistent
Sequential consistency

- $so \subseteq vis$ and $vis$ is total
- $vis \subseteq ar \iff$ can equivalently require $so \subseteq vis = ar$
- Every operation sees the effect of all operations preceding it in $vis$
- Like the original definition with $to = vis = ar$
Dekker example

\[
x.\text{write}(1) \\
\text{vis} \quad \text{so} \\
y.\text{read}(): 0
\]

\[
y.\text{write}(1) \\
\text{so} \quad \text{vis} \\
x.\text{read}(): 0
\]
Dekker example

\[\begin{align*}
x &. \text{write}(1) \\
y &. \text{read}(): 0 \\
\text{vis} & \downarrow \text{so} \\
y &. \text{read}(): 0 \\
x &. \text{read}(): 0 \\
\text{so} & \downarrow \text{vis} \\
y &. \text{write}(1) \\
\text{vis} & \downarrow \text{ar, vis}
\end{align*}\]
Dekker example

x.write(1)  \rightarrow ar, vis  \rightarrow y.write(1)

vis \quad so  \quad so  \quad vis

y.read(): 0  \rightarrow ar, vis  \rightarrow x.read(): 0
Dekker example

No execution with such history
Consistency zoo

- Eventual consistency

- Session guarantees: **Dekker, IRIW, causality violation**
  \[ so \subseteq vis, \text{ vis}; \ so \subseteq \text{ vis} \]

- Causal consistency: **Dekker, IRIW**
  \[ (so \cup \text{ vis})^+ \subseteq \text{ vis} \]

- Prefix consistency: **Dekker**
  \[ \text{ar; (vis \setminus so)} \subseteq \text{ vis} \]

- Sequential consistency
  \[ \text{vis} = \text{ar} \]
Shared-memory models

• Sequential consistency first proposed in the context of shared memory (1979)

• Processors and languages don’t provide sequential consistency: weak memory models, due to processor and compiler optimisations

• Our specifications similar to weak memory model definitions

• Consistency axioms for last-writer-wins registers ~ shared-memory models
Consistency Zoo

• Eventual consistency

• Session guarantees: Dekker, IRIW, causality violation
  \[ so \subseteq vis, vis; so \subseteq vis \]

• Causal consistency: Dekker, IRIW
  \[ (so \cup vis)^+ \subseteq vis \]

• Prefix consistency: Dekker
  \[ ar; (vis \setminus so) \subseteq vis \]

• Sequential consistency
  \[ vis = ar \]
Theoretical results

- Eventual consistency
- Session guarantees
  \[ \text{so} \subseteq \text{vis}, \text{vis}; \text{so} \subseteq \text{vis} \]
- Causal consistency
  \[ (\text{so} \cup \text{vis})^+ \subseteq \text{vis} \]
- Prefix consistency
  \[ \text{ar}; (\text{vis} \setminus \text{so}) \subseteq \text{vis} \]
- Sequential consistency
  \[ \text{vis} = \text{ar} \]

- What's the best we can do while staying available under network partitionings?
- Causal consistency is a strongest such model [Attiya et al., 2015]
Theoretical results

- Eventual consistency
- Session guarantees
  \[ so \subseteq \text{vis}, \text{vis}; so \subseteq \text{vis} \]
- Causal consistency
  \[ (so \cup \text{vis})^+ \subseteq \text{vis} \]
- Prefix consistency
  \[ \text{ar}; (\text{vis} \setminus so) \subseteq \text{vis} \]
- Sequential consistency
  \[ \text{vis} = \text{ar} \]

- What's the best we can do while staying available under network partitionings?
- Causal consistency is the strongest such model [Attiya et al., 2015]
Theoretical results

- Eventual consistency
- Session guarantees
  $so \subseteq vis, vis; so \subseteq vis$
- Causal consistency
  $(so \cup vis)^+ \subseteq vis$
- Prefix consistency
  $ar; (vis \setminus so) \subseteq vis$
- Sequential consistency
  $vis = ar$

Terms and conditions apply:

- What's the best we can do while staying available under network partitionings?
- Causal consistency is a strongest such model [Attiya et al., 2015]

- for a certain version of CC and a certain class of implementations
- a strongest model: cannot be strengthened, but can be other alternative incomparable models
Theoretical results

- Application of eventual consistency - collaborative editing: Google Docs, Office Online

- At the core: list data type (of formatted characters)

- List data type has an inherently high metadata overhead: can't discard a character when deleting it from a Google Docs document! [Attiya et al., 2016]

- Discarding may allow previously deleted elements to reappear
Determining the right level of consistency
Application correctness

- Does an application satisfy a particular correctness property?

  *Integrity invariants:* account balance is non-negative

- Is an application robust against a particular consistency model?

  *Application behaves the same as when using a strongly consistent database*
Application correctness

• Does an application satisfy a particular correctness property?

  Integrity invariants: account balance is non-negative

• Is an application robust against a particular consistency model?

  Application behaves the same as when using a strongly consistent database
Challenge

Vanilla weak consistency often too weak to preserve correctness

Need to strengthen consistency in parts of the application
Deposits

\[
\boxed{\text{val}} \quad \boxed{\text{add(100)}} \quad \boxed{\text{val}}
\]

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

\[
[\text{add(100)}]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100)
\]
Withdrawals

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

$\sigma$

$\begin{align*}
\text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma')
\end{align*}$
Withdrawals

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

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

balance = 100
withdraw(100) : ✔

\[
\llbracket \text{withdraw}(100) \rrbracket_{\text{eff}}(\sigma) = \\
\text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma')
\]
\[
\begin{align*}
\llbracket \text{withdraw}(100) \rrbracket_{\text{eff}}(\sigma) &= \\
&= \text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma')
\end{align*}
\]
\[
\text{withdraw(100)} : \checkmark
\]

\[
\lambda \sigma'. \sigma' - 100
\]

\[
[\text{withdraw(100)}]_{\text{eff}}(\sigma) = \begin{cases} 
\lambda \sigma'. \sigma' - 100 & \text{if } \sigma \geq 100 \\
\lambda \sigma'. \sigma' & \text{else}
\end{cases}
\]
\begin{align*}
\text{balance} &= 100 \\
\text{withdraw}(100) &: \checkmark \\
\text{balance} &= 0 \\
\text{add}(100) &: \checkmark \\
\text{balance} &= 100 \\
\end{align*}
balance = 100

withdraw(100) : ✔️

λσ'. σ' - 100

balance = 0

balance = 0

withdraw(100) : ✔️

balance = -100

balance = 100

add(100) : ✔️

Tune consistency:
- Withdrawals strongly consistent
- Deposits eventually consistent
Strengthening consistency

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

- Symmetric conflict relation on operations:
  \( \otimes \subseteq Op \times Op \), e.g., withdraw \( \otimes \) withdraw

- Conflicting operations cannot be causally independent:
  \[
  \forall e, f \in E. \ op(e) \otimes op(f) \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e
  \]
Strengthening consistency

- Symmetric conflict relation on operations:
  \[ \bowtie \subseteq \text{Op} \times \text{Op}, \text{e.g., withdraw} \bowtie \text{withdraw} \]

- Conflicting operations cannot be causally independent:
  \[ \forall e, f \in E. \text{op}(e) \bowtie \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e \]
Strengthening consistency

- Symmetric conflict relation on operations: \( \bowtie \subseteq \text{Op} \times \text{Op} \), e.g., withdraw \( \bowtie \) withdraw

- Conflicting operations cannot be causally independent:
  \[ \forall e, f \in E. \text{op}(e) \bowtie \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e \]

- No constraints on additions: \( \neg (\text{add} \bowtie \text{op}) \)
Strengthening consistency

- Implementation requires replicas executing `withdraw()` to synchronise
- `add()` doesn't need synchronisation
balance = 100
withdraw(100) : ✔

withdraw \times withdraw: as if withdraw grabs an exclusive lock (mutex) on the account
`withdraw × withdraw`: as if `withdraw` grabs an exclusive lock (mutex) on the account

- `balance = 100`
- `withdraw(100) : ✔`

- `balance = 100`
  - `withdraw(100) : ?`
withdraw \times withdraw: as if withdraw grabs an exclusive lock (mutex) on the account

balance = 100
withdraw(100) : ✓

balance = 100
withdraw(100) : ?
withdraw ⋈ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account

Acquiring the lock requires bringing all operations the replica holding it knows about
withdraw ✧ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account
withdraw $\not\land$ withdraw: as if withdraw grabs an exclusive lock (mutex) on the account

balance = 100
withdraw(100) : ✔
withdraw $\not\land$ withdraw

balance = 100
withdraw(100) : ✗

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

¬(add $\not\land$ op): no locks, so no synchronisation
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

• Stronger operations require synchronisation between replicas

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

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

• Reason about all possible abstract executions?
  
  ‣ Abstract from some of implementation details, but still describe behaviour of the whole system
  
  ‣ Number of possible executions is exponential: e.g., choices of \( vis \) = order of message deliveries

• Need **verification techniques** that limit the exponential blow-up
Verification problem

Given

• a set of operations: `withdraw()`, `deposit()`, ...

• a conflict relation: `withdraw ⋈ withdraw`

Do the operations always preserve a given integrity invariant?

\[ I = (balance \geq 0) \]
Verification problem

Given

• a set of operations: `withdraw()`, `deposit()`, ...

• a conflict relation: `withdraw ⋈ withdraw`

Do the operations always preserve a given integrity invariant?

I = (balance ≥ 0)

Later: operations ➔ whole transactions
σ ∈ I

Assume invariant holds

op

Check it’s preserved after executing op

Single check: no state-space explosion from concurrency
Effect applied in a different state!
\[ \text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else (} \lambda \sigma'. \sigma' ) \]
\[ [\text{op}]_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety**: \( f(\sigma) \) preserves \( I \) when executed in any state satisfying \( P \): \( \{I \land P\} f(\sigma) \{I\} \)
\[
\sigma \in I \\
\text{op} \\
\llbracket \text{op} \rrbracket_{\text{eff}}(\sigma) \\
\sigma' \\
\llbracket \text{op} \rrbracket_{\text{eff}}(\sigma)(\sigma') \in I \?
\]

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

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

\[
\{ \text{bal} \geq 0 \land \text{bal} \geq 100 \} \text{ bal := bal-100 } \{ \text{bal} \geq 0 \}
\]
\[ \left[ \text{op} \right]_{\text{eff}}(\sigma)(\sigma') \in I? \]

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

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

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

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

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

\[ \{ \text{bal} \geq 0 \land \text{bal} \geq 100 \} \quad \text{bal := bal-100} \quad \{ \text{bal} \geq 0 \} \]
\[
\boxed{\text{Effector safety: } f(\sigma) \text{ preserves } I \text{ when executed in any state satisfying } P: \{I \land P\} f(\sigma) \{I\}}
\]

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

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

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

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

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

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

\{bal \geq 0 \land bal \geq 100\} \quad \text{bal} := \text{bal}-100 \quad \{bal \geq 0\}$
\[ \muop \sigma' \in I \] is an effector.

\[ \muop \rho(\sigma) \] is computed as:

\[ \muop \rho(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

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

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

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

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
$\sigma \in I$

$\text{op} \xrightarrow{\text{eff}(\sigma)} \sigma'$

$P(\sigma')$?
σ ∈ I

op

σ'
P(σ')?

[op]_{eff}(σ)

op’s causal dependencies
\[ \sigma \in I \]

- Causal consistency \( \Rightarrow \) receive op’s causal dependencies before receiving op

\[ \sigma \]

\[ \sigma' \]

\[ P(\sigma')? \]
- 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 $\sigma' = (f; g; ...)(\sigma)$
• 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: $\{P\} f \{P\}$

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

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

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

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

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

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

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

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

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

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

Only requires checking each pair of operations: no exponential explosion!
Can infer the conflict relation $\bowtie$: $\text{op}_1 \bowtie \text{op}_2$ if the precondition of $\text{op}_1$ unstable under the effector of $\text{op}_2$.

Pre of withdraw under effector of add:

\[ \{ \text{bal} \geq 100 \} \text{ bal} := \text{bal} + 100 \{ \text{bal} \geq 100 \} \] ✔, no $\bowtie$
Can infer the conflict relation $\bowtie: op_1 \bowtie op_2$ if the precondition of $op_1$ unstable under the effector of $op_2$

Pre of withdraw under effector of withdraw:

$\{bal \geq 100\}$ bal := bal-100 $\{bal \geq 100\}$ $\times$, need $\bowtie$
Correct Eventual Consistency Tool

- Developed by Sreeja Nair (UPMC, Paris)

- Model application in a domain-specific language, including replicated data type libraries

- Model compiled into a Boogie program encoding the conditions of the proof rule

- Discharged using SMT

- Automatically infers a conflict relation

https://github.com/LightKone/correct-eventual-consistency-tool
Demo
Transactions
Transactions

- Fundamental abstraction in databases
- Allow clients to group operations to be processed indivisibly
- Provided by virtually any single-node SQL database
- NoSQL data stores: starting to reappear
set.add(photo)
reg.write(post)

so

set.read() \ni

\exists \text{ photo}

so

reg.read() : \emptyset
set.add(photo)

so

reg.write(post)

set.read() \ni\in photo

so

reg.read() : \emptyset \times
Causal consistency isn't enough
```
set.add(photo)  
reg.write(post)  

set.read() \ni  photo

reg.read() : post
```
set.add(photo)

so

reg.write(post)

set.read() \ni\in\ photo

so

reg.read() : post
- Consistency model = set of histories \( (E, so, \sim) \)
• Consistency model = set of histories \((E, so, \sim)\)

• \(\sim\): equivalence relation that groups events from the same transaction: transitive, symmetric, reflexive
Consistency model = set of histories \((E, \text{so}, \sim)\)

\(\sim\): equivalence relation that groups events from the same transaction: transitive, symmetric, reflexive

For simplicity, assume every transaction completes
• Consistency model = set of histories \((E, so, \sim)\)

• \(\sim\): equivalence relation that groups events from the same transaction: transitive, symmetric, reflexive

• For simplicity, assume every transaction completes

• Transaction \(T\): equivalence class of events of \(\sim\)
set.add(photo)
so
reg.write(post)
so
set.add(photo2)
...
A session is a sequence of transactions: events from the same transaction contiguous in so
\[ \forall e, f, g \in E. \ e \overset{so}{\rightarrow} f \overset{so}{\rightarrow} g \land e \sim g \]
\[ \implies e \sim f \sim g \]
Strongly consistent transactions

Sequential consistency ~ serializability
Serializability operationally

• Server with a single copy of all objects
• Clients send txs to the server and wait for a reply
• Server processes txs atomically in the receipt order
Serializability operationally

- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order
Server with a single copy of all objects

Clients send txs to the server and wait for a reply

Server processes txs atomically in the receipt order
Serializability operationally

- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order
Serializability operationally

- Server with a single copy of all objects
- Clients send txs to the server and wait for a reply
- Server processes txs atomically in the receipt order

Diagram:
- set.add(photo)
- reg.write(post)
- (ok, ok)
- tx1
- tx2

set, reg

- set.read()
- reg.read()

- ({photo}, post)
Serializability operationally

Serializability = \{H \mid \exists \text{ execution with history } H \text{ produced by the abstract implementation}\}
Sequential consistency

(E,so) | ∃ total order to. (E, so, to) satisfies:

1. so ⊆ to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to
Serializability

\[(E, so, \sim) \mid \exists \text{ total order } to. (E, so, \sim, to) \text{ satisfies:} \]

1. so $\subseteq$ to

2. The return value of each operation in E is computed from a state obtained by executing all operations on the same object preceding it in to

3. Operations from the same transaction are contiguous in to
set.add(photo)

reg.write(post)

set.add(photo2)

set.read() ∋ photo

reg.read() : post

Operations from the same transaction are contiguous in to
set.add(photo)
reg.write(post)

set.add(photo2)
...

set.read() \( \ni \) photo
reg.read() : post

Operations from the same transaction are contiguous in to.
Operations from the same transaction are contiguous in time

Induces a total \( \text{to/\sim} \) on whole tx
Weakening consistency

• Even single-node databases don't provide serializability either by default or at all: read committed, snapshot isolation, ...
• Even single-node databases don't provide serializability either by default or at all: read committed, snapshot isolation, ...

• To better exploit single-node parallelism
Eventually consistent transactions

- Single-node consistency models also applicable in distributed setting

- But many still require some synchronisation between replicas: unavailability, high latency

- Want eventually consistent transactions: always available, low latency

- Preserve some aspects of the invisibility abstraction
System model recap

• Database system consisting of multiple reliable replicas

• Each replica stores a copy of all objects of replicated data types

• Replicas can communicate via asynchronous reliable channels
A client connects to a replica and issues transactions

High availability: the transaction commits immediately, without communication with other replicas, no aborts!
• A client connects to a replica and issues transactions

• **High availability:** the transaction **commits** immediately, without communication with other replicas, no **aborts**!

• Replica processes transactions **sequentially:** anomalies arising from single-node concurrency covered by the absence of inter-node synchronisation
- A client connects to a replica and issues transactions

- **High availability:** the transaction commits immediately, without communication with other replicas, no aborts!

- Replica processes transactions sequentially: anomalies arising from single-node concurrency covered by the absence of inter-node synchronisation

- **Reads are indivisible:** access a fixed snapshot of the database (plus own writes)
Upon commit: send the effectors of all tx operations to other replicas together

x.write(*post*)
y.write(*comment*)
x.read : *post*
Upon commit: send the effectors of all tx operations to other replicas together

Receive in between txs: incorporate all the updates together
Upon commit: send the effectors of all tx operations to other replicas **together**

Receive in between txs: incorporate all the updates **together**

- Writes are indivisible
- Reads are indivisible
- Reads+writes: no!
Reads/writes indivisibility

set.add(photo) → reg.write(post)

set.read() ∈ photo

so

reg.read() : post
No reads+writes indivisibility

reg: last-writer-wins register, initially 0

v = reg.read()  // 0
reg.write(v+1)  // 1

v = reg.read()  // 0
reg.write(v+1)  // 1
No reads+writes indivisibility

`reg`: last-writer-wins register, initially 0

\[
\begin{align*}
v &= \text{reg.read()} \quad \text{// 0} \\
\text{reg.write}(v+1) &= \text{// 1} \\
\text{reg.read()} &= \text{// 1}
\end{align*}
\]

\[
\begin{align*}
v &= \text{reg.read()} \quad \text{// 0} \\
\text{reg.write}(v+1) &= \text{// 1} \\
\text{reg.read()} &= \text{// 1}
\end{align*}
\]
No reads+writes indivisibility

reg: last-writer-wins register, initially 0

\[
\begin{align*}
v &= \text{reg.read()} \quad \text{// 0} \\
\text{reg.write}(v+1) &= \text{// 1} \\
\end{align*}
\]

\[
\begin{align*}
v &= \text{reg.read()} \quad \text{// 0} \\
\text{reg.write}(v+1) &= \text{// 1} \\
\end{align*}
\]

Lost update anomaly
Use appropriate data type

counter: replicated counter, accumulates increments initially 0

counter.add(1)

counter.add(1)

counter.read() : 2
Operational specification

- **Eventual consistency with transactions** = the set of all histories produced by arbitrary client interactions with the data type implementations (with any allowed message deliveries)

- Implies **quiescent consistency**: if no new updates are made to the database, then replicas will eventually converge to the same state
Axiomatic specification

- Serializability: operations from the same transaction are contiguous in the total order to

- Approach: require the same of vis and ar
Operations from the same transaction are contiguous in $\text{to}$:

$$\forall e, f, e', f'. \ e \sim f \land e' \sim e \xrightarrow{\text{to}} f \sim f' \xrightarrow{\text{to}} e' \xrightarrow{\text{to}} f'$$
Serializability: \((E, \text{so}, \sim, \text{to})\)

Operations from the same transaction are contiguous in \text{to}:

\[ \forall e, f, e', f'. \ e \sim f \land e' \sim e \ \overset{\text{to}}{\longrightarrow} \ f \sim f' \ \iff \ e' \ \overset{\text{to}}{\longrightarrow} \ f' \]
Serializability: \((E, \text{so}, \sim, \text{to})\)

Operations from the same transaction are contiguous in \(\text{to}\):

\[
\forall e, f, e', f'.\ e \sim f \land e' \sim e \implies f \sim f' \implies e' \implies f'
\]
Serializability: \((E, \text{so}, \sim, \text{to})\)

Operations from the same transaction are contiguous in \text{to}:

\[
\forall e, f, e', f'.\ e \sim f \land e' \sim e \quad \overset{\text{to}}{\longrightarrow} \quad f \sim f' \quad \iff \quad e' \quad \overset{\text{to}}{\longrightarrow} \quad f'
\]
Operations from the same transaction are contiguous in \( \text{to} \):

\[
\forall e, f, e', f'. \left( e \not\sim f \land e' \sim e \right) \rightarrow f \sim f' \implies e' \rightarrow f'
\]
Serializability: \((E, \text{ so}, \sim, \text{ to})\)

Operations from the same transaction are contiguous in \text{to}:

\[
\forall e, f, e', f'. e \sim f \land e' \sim e \xrightarrow{\text{to}} f \sim f' \implies e' \xrightarrow{\text{to}} f'
\]

to treats events in a transaction uniformly
Execution: \((E, \text{so}, \sim, \text{vis}, \text{ar})\)

- `set.add(photo)`
- `reg.write(post)`

- `set.read() \ni \exists\, \text{photo}`
- `reg.read() : \emptyset`

**vis, ar treat events in a transaction uniformly:**

\[
\forall e, f, e', f'.\ e \not\sim f \land e' \sim e \ \xrightarrow{\text{vis}} \ f \sim f' \ \xrightarrow{} \ e' \ \xrightarrow{\text{vis}} \ f' \\
\forall e, f, e', f'.\ e \not\sim f \land e' \sim e \ \xrightarrow{\text{ar}} \ f \sim f' \ \xrightarrow{} \ e' \ \xrightarrow{\text{ar}} \ f'
\]
Execution: $(E, \text{so}, \sim, \text{vis}, \text{ar})$

- set.add(\text{photo})
- reg.write(\text{post})
- set.read() $\ni$ \text{photo}
- reg.read() : $\emptyset$

\text{vis, ar} treat events in a transaction uniformly:

\begin{align*}
\forall e, f, e', f'. e &\not\sim f \land e' \sim e \quad \overset{\text{vis}}{\Longrightarrow} \quad f \sim f' \quad \Longrightarrow \quad e' \quad \overset{\text{vis}}{\Longrightarrow} \quad f' \\
\forall e, f, e', f'. e &\not\sim f \land e' \sim e \quad \overset{\text{ar}}{\Longrightarrow} \quad f \sim f' \quad \Longrightarrow \quad e' \quad \overset{\text{ar}}{\Longrightarrow} \quad f'
\end{align*}
vis, ar treat events in a transaction uniformly:

\[ \forall e, f, e', f'. e \sim f \land e' \sim e \implies f \sim f' \implies e' \implies f' \]

\[ \forall e, f, e', f'. e \sim f \land e' \sim e \implies f \sim f' \implies e' \implies f' \]
Execution: \((E, \text{ so}, \sim, \text{ vis}, \text{ ar})\)

**vis, ar** treat events in a transaction uniformly:

\[\forall e, f, e', f'. \ e \not\sim f \land e' \sim e \quad \text{vis} \quad f \sim f' \quad \Longrightarrow \quad e' \quad \text{vis} \quad f'\]

\[\forall e, f, e', f'. \ e \not\sim f \land e' \sim e \quad \text{ar} \quad f \sim f' \quad \Longrightarrow \quad e' \quad \text{ar} \quad f'\]
Execution: \((E, \text{so}, \sim, \text{vis}, \text{ar})\)

\[
\begin{align*}
\text{set.add(photo)} & \quad \text{vis} \quad \text{set.read()} \ni \text{photo} \\
\text{reg.write(post)} & \quad \text{so} \quad \text{reg.read()} : \text{post}
\end{align*}
\]

\text{vis, ar} \text{ treat events in a transaction uniformly:}

\[
\forall e, f, e', f'. e \not\sim f \land e' \sim e \quad \xrightarrow{\text{vis}} \quad f \sim f' \quad \implies \quad e' \quad \xrightarrow{\text{vis}} \quad f'
\]

\[
\forall e, f, e', f'. e \not\sim f \land e' \sim e \quad \xrightarrow{\text{ar}} \quad f \sim f' \quad \implies \quad e' \quad \xrightarrow{\text{ar}} \quad f'
\]
Execution: \((E, \text{so}, \sim, \text{vis}, \text{ar})\)

\[ \text{set.add(photo)} \quad \text{vis} \quad \text{set.read()} \ni \text{photo} \]

\[ \text{reg.write(post)} \quad \text{vis} \quad \text{reg.read()} : \text{post} \]

\text{vis, ar induce acyclic vis/\sim, ar/\sim on whole txs:}

\[ T \overset{\text{vis/\sim}}{\longrightarrow} S \iff \exists e \in T, f \in S. e \overset{\text{vis}}{\longrightarrow} f \]

\[ T \overset{\text{ar/\sim}}{\longrightarrow} S \iff \exists e \in T, f \in S. e \overset{\text{ar}}{\longrightarrow} f \]
Eventually consistent transactions

The set of histories \((E, \text{so}, \sim)\) such that for some \(\text{vis}, \text{ar}\):

- Return values consistent with data type specs:
  \[ \forall e \in E. \text{rval}(e) = F_{\text{type}(\text{obj}(e))}(\text{context}(e)) \]

- No causal cycles: \(\text{so} \cup \text{vis}\) is acyclic

- Eventual visibility:
  \[ \forall e \in E. e \overset{\text{vis}}{\rightarrow} f \text{ for all but finitely many } f \in E \]

- Transaction indivisibility:
  \[ \forall e, f, e', f'. e \not\sim f \land e' \sim e \overset{\text{vis}}{\rightarrow} f \sim f' \implies e' \overset{\text{vis}}{\rightarrow} f' \]
  \[ \forall e, f, e', f'. e \not\sim f \land e' \sim e \overset{\text{ar}}{\rightarrow} f \sim f' \implies e' \overset{\text{ar}}{\rightarrow} f' \]
Define transactional variants of other consistency models by just adding prior axioms

**Serializable:** $\text{vis} = \text{ar}$

- Return values consistent with data type specs:
  $\forall e \in E. \text{rval}(e) = F_{\text{type(obj(e))}}(\text{context(e)})$

- No causal cycles: $\text{so} \cup \text{vis}$ is acyclic

- Eventual visibility:
  $\forall e \in E. e \xrightarrow{\text{vis}} f$ for all but finitely many $f \in E$

- Transaction indivisibility:
  $\forall e, f, e', f'. e \sim f \land e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f'$
  $\forall e, f, e', f'. e \sim f \land e' \sim e \xrightarrow{\text{ar}} f \sim f' \implies e' \xrightarrow{\text{ar}} f'$
Session guarantees

```
set.add(photo)
reg.write(post)
reg.read(): ?
```

Transactions in the same session only accumulate information

\[
\text{so} \subseteq \text{vis}
\]
Session guarantees

\[ \text{set.add(photo)} \]
\[ \rightarrow \text{so} \]
\[ \text{reg.write(post)} \]
\[ \rightarrow \text{so} \quad \text{and} \quad \text{vis} \]
\[ \text{reg.read(): post} \]

Transactions in the same session only accumulate information

\[ \text{so} \subseteq \text{vis} \]
Causal consistency

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

\[
\text{set.add(}\text{photo}\text{)} \\
\text{reg.write(}\text{post}\text{)} \\
\text{so} \\
\text{(so }\cup\text{ vis)}^+ \subseteq \text{vis}
\]
Causal consistency

(set.add(photo) → so → reg.write(post) → so → vis → reg.read(): post → so → vis) →

(reg2.write(comment) → vis → (so ∪ vis)⁺ ⊆ vis)
Causal consistency

\[
\text{set.add(} \text{photo)} \\
\text{reg.write(} \text{post)} \\
\text{vis} \\
\text{vis} \\
\text{reg.read(): post} \\
\text{vis} \\
\text{reg2.write(} \text{comment)} \\
\text{vis} \\
\text{vis} \\
\text{set.read(): ?} \\
\text{vis} \\
\text{reg2.read(): comment}
\]

\[
(s_0 \cup \text{vis})^+ \subseteq \text{vis}
\]
Causal consistency

```
set.add(photo)

reg.write(post)

so

vis

reg.read(): post

reg2.write(comment)

so

vis

(vis ∪ so)⁺ ⊆ vis

set.read(): ?

so

vis

reg2.read(): comment
```
Causal consistency

()`so ∪ vis)^+ ⊆ vis`
Causal consistency

- `set.add(photo)`
- `reg.write(post)`
- `reg.read(): post`
- `reg2.write(comment)`
- `set.read() ∋ photo`
- `(so ∪ vis)^+ ⊆ vis`
- `reg2.read(): comment`

`vis` and `so` are annotations on the transitions.
Concurrent withdrawals

c: counter with decrements, initially 100

\[
v = c.\text{read()}
\]
\[
\text{if } (v \geq 100)
\]
\[
c.\text{subtract(100)}
\]

\[
v = c.\text{read()}
\]
\[
\text{if } (v \geq 100)
\]
\[
c.\text{subtract(100)}
\]
Concurrent withdrawals

c: counter with decrements, initially 100

\[
\begin{align*}
\text{v} &= \text{c.read()} \quad // \quad 100 \\
\text{if } (\text{v} \geq 100) &\quad \rightarrow \quad \text{so} \\
\text{c.subtract(100)} &= \quad // \quad 0
\end{align*}
\]
Concurrent withdrawals

\( c \): counter with decrements, initially 100

\[
\begin{align*}
v &= c.\text{read}() \quad \text{// 100} \\
\text{if } (v \geq 100) &\quad \text{so} \\
c.\text{subtract}(100) &\quad \text{// 0}
\end{align*}
\]

Both transactions decremented successfully - synchronisation needed!
Recap: strengthening consistency

- Baseline model: causal consistency
- Symmetric conflict relation on operations: \( \otimes \subseteq \text{Op} \times \text{Op} \), e.g., withdraw \( \otimes \) withdraw
- Conflicting operations cannot be causally independent:
  \[
  \forall e, f \in E. \ \text{op}(e) \otimes \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e
  \]
Recap: strengthening consistency

- Baseline model: causal consistency
- Symmetric conflict relation on operations: \( \otimes \subseteq \text{Op} \times \text{Op} \), e.g., withdraw \( \otimes \) withdraw
- Conflicting operations cannot be causally independent:
  \[ \forall e, f \in E. \text{op}(e) \otimes \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e \]
Strengthening transactions

- Baseline model: causal consistency
- Symmetric conflict relation on operations: \( \Join \subseteq \text{Op} \times \text{Op} \), e.g., \( \text{subtract} \Join \text{subtract} \)
- Conflicting operations cannot be causally independent:
  \[
  \forall e, f \in E. \text{op}(e) \Join \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e
  \]
Strengthening transactions

- Baseline model: causal consistency
- Symmetric conflict relation on operations: \( \bigotimes \subseteq \text{Op} \times \text{Op} \), e.g., \text{subtract} \( \bigotimes \) \text{subtract}
- Conflicting operations cannot be causally independent:
  \[ \forall e, f \in E. \text{op}(e) \bigotimes \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e \]
Strengthening transactions

- Baseline model: causal consistency
- Symmetric conflict relation on operations: \( \otimes \subseteq \text{Op} \times \text{Op} \), e.g., \( \text{subtract} \otimes \text{subtract} \)
- Conflicting operations cannot be causally independent:

\[
\forall e, f \in E. \text{op}(e) \otimes \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e
\]
Strengthening transactions

- Baseline model: causal consistency

- Symmetric conflict relation on operations: \( \Join \subseteq \text{Op} \times \text{Op} \), e.g., \( \text{subtract} \Join \text{subtract} \)

- Conflicting operations cannot be causally independent:
  \[
  \forall e, f \in E. \, \text{op}(e) \Join \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e
  \]
Strengthening transactions

- Baseline model: causal consistency
- Symmetric conflict relation on operations: △ ⊆ Op × Op, e.g., subtract △ subtract
- Conflicting operations cannot be causally independent:
  \[ \forall e, f \in E. \text{op}(e) \triangle \text{op}(f) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e \]
Recap: implementation

• withdraw ⋈ withdraw: as if withdraw grabs an exclusive lock on the account

• Acquiring the lock requires bringing all operations the replica holding it knows about

\[ \text{c.withdraw}(100) : \checkmark \quad \text{c.withdraw}(100) : \mathbf{?} \]
Recap: implementation

- `withdraw ⋈ withdraw`: as if `withdraw` grabs an exclusive lock on the account
- Acquiring the lock requires bringing all operations the replica holding it knows about

\[ c.\text{withdraw}(100) : \checkmark \rightarrow c.\text{withdraw}(100) : \? \]
Recap: implementation

• **withdraw ⋈ withdraw**: as if withdraw grabs an exclusive lock on the account

• Acquiring the lock requires bringing all operations the replica holding it knows about
Implementation for transactions

```java
v = c.read()  // 100
if (v >= 100)
c.subtract(100)  // 0
```
Implementation for transactions

\[
v = c.read() \quad // \quad 100
\]
\[
\text{if } (v \geq 100)
\]
\[
c.subtract(100) \quad // \quad 0
\]
Implementation for transactions

\[ v = c.\text{read()} \quad // \quad 100 \]
if \( v \geq 100 \)
\[ c.\text{subtract}(100) \quad // \quad 0 \]

\[ v = c.\text{read()} \quad // \quad 100 \]
Implementation for transactions

\[
\begin{align*}
\text{v} &= \text{c.read()} \quad // \ 100 \\
&\text{if } (\text{v} \geq 100) \\
&\text{c.subtract}(100) \quad // \ 0
\end{align*}
\]

\[
\begin{align*}
\text{v} &= \text{c.read()} \quad // \ 100 \\
&\text{if } (\text{v} \geq 100)
\end{align*}
\]
Implementation for transactions

\[
\begin{align*}
v &= \text{c.read()} \quad // 100 \\
\text{if } (v \geq 100) \\
\text{c.subtract}(100) \quad // 0 \\
\end{align*}
\]

\[
\begin{align*}
v &= \text{c.read()} \quad // 100 \\
\text{if } (v \geq 100) \\
\text{c.subtract}(100)
\end{align*}
\]
Implementation for transactions

\[ v = c.\text{read()} \quad \text{// 100} \]
\[ \text{if } (v \geq 100) \]
\[ c.\text{subtract}(100) \quad \text{// 0} \]

\[ v = c.\text{read()} \quad \text{// 100} \]
\[ \text{if } (v \geq 100) \]
\[ c.\text{subtract}(100) \]

- Need to incorporate the effector of the previous transaction
Implementation for transactions

v = c.read()  // 100
if (v >= 100)
c.subtract(100)  // 0

v = c.read()  // 100
if (v >= 100)
c.subtract(100)

• Need to incorporate the effector of the previous transaction
• Recall: transactions execute on a fixed snapshot
• Too late: effectors from other replicas only get applied in-between transactions
• Have to abort the transaction and re-execute it
Implementation for transactions

• Need to incorporate the effector of the previous transaction
• Recall: transactions execute on a fixed snapshot
• Too late: effectors from other replicas only get applied in-between transactions
• Have to abort the transaction and re-execute it

```java
v = c.read()   // 100
if (v ≥ 100)
c.subtract(100)   // 0
```
Implementation for transactions

Need to incorporate the effector of the previous transaction

Recall: transactions execute on a fixed snapshot

Too late: effectors from other replicas only get applied in-between transactions

Have to **abort** the transaction and re-execute it

```python
v = c.read()  # 100
if (v >= 100):
    c.subtract(100)  # 0
```
Chosing ⋈

- Want to choose ⋈ to preserve application invariants
- Previous proof rule for checking invariants applies
- Instead of an effector of a single operation, consider a sequential composition of effectors of all operations in a transaction
- Can also fix ⋈ so that it's easier to program: new consistency models, disallowing some classes of anomalies
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

\[ \forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e \]
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[\forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e\]

- No overdrafts:
  
  \[v = \text{c.read()} \land 100\]
  \[\text{if } (v \geq 100) \downarrow \text{so}\]
  \[\text{c.subtract(100)} \land 0\]
  
  \[v = \text{c.read()} \land 100\]
  \[\text{if } (v \geq 100) \downarrow \text{so}\]
  \[\text{c.subtract(100)} \land 0\]
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[
  \forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \implies e \text{ vis} f \lor f \text{ vis} e
  \]

- No overdrafts:
  
  ```plaintext
  v = c.read() // 100
  if (v \geq 100) so
  c.subtract(100) // 0
  
  v = c.read() // 100
  if (v \geq 100) so
  c.subtract(100) // 0
  ```
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[ \forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e \]

- No overdrafts:

```
v = c.read() // 100
if (v \geq 100) so
c.subtract(100) // 0
```
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

\[
\forall e, f \in E. \text{obj}(e) = \text{obj}(f) \land \text{update}(\text{op}(e)) \land \text{update}(\text{op}(f)) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e
\]

- No overdrafts:

```plaintext
v = c.read()  // 100
if (v ≥ 100)  so
  c.subtract(100)  // 0
```

```plaintext
v = c.read()  // 100
if (v ≥ 100)  so
  c.subtract(100)  // 0
```
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[ \forall e, f \in E. \text{obj}(e) = \text{obj}(f) \land \text{update}(\text{op}(e)) \land \text{update}(\text{op}(f)) \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e \]

- No lost updates:

\[
\begin{align*}
v &= \text{reg.read()} \quad // 0 \\
&\downarrow_{\text{so}} \\
\text{reg.write}(v+1) \quad // 1
\end{align*}
\]

\[
\begin{align*}
v &= \text{reg.read()} \quad // 0 \\
&\downarrow_{\text{so}} \\
\text{reg.write}(v+1) \quad // 1
\end{align*}
\]
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[ \forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e \]

- No lost updates:

\[
\begin{array}{ll}
v = \text{reg.read()} & // 0 \\
\text{reg.write}(v+1) & // 1 \\
\end{array}
\]

\[
\begin{array}{ll}
v = \text{reg.read()} & // 0 \\
\text{reg.write}(v+1) & // 1 \\
\end{array}
\]
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:

\[ \forall e, f \in E. \ obj(e) = obj(f) \land \ update(op(e)) \land \ update(op(f)) \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e \]

- No lost updates:

\[
\begin{align*}
v &= \text{reg.read()} \quad \text{// 0} \\
\text{reg.write}(v+1) &\quad \text{// 1}
\end{align*}
\]

\[
\begin{align*}
v &= \text{reg.read()} \quad \text{// 0} \\
\text{reg.write}(v+1) &\quad \text{// 1}
\end{align*}
\]
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[
  \forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \\
  \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e
  \]

- No lost updates:

```plaintext
v = reg.read() // 0
reg.write(v+1) // 1

v = reg.read() // 0
reg.write(v+1) // 1
```
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[ \forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e \]

- Updates on different accounts can go in parallel:

\[
\begin{align*}
v &= \text{reg.read()} \quad // 0 \\
\downarrow \text{so} \\
\text{reg.write}(v+1) \quad // 1 \\
\end{align*}
\]

\[
\begin{align*}
v' &= \text{reg'.read()} \quad // 0 \\
\downarrow \text{so} \\
\text{reg'.write}(v+1) \quad // 1 \\
\end{align*}
\]
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[
  \forall e, f \in E. \ obj(e) = obj(f) \land \text{update}(\text{op}(e)) \land \text{update}(\text{op}(f)) \implies e \overset{\text{vis}}{\longrightarrow} f \lor f \overset{\text{vis}}{\longrightarrow} e
  \]

- Visibility totally orders transactions updating the same object \implies don't need replicated data types, don't need ar

```
set.add(1) \overset{\text{vis}}{\rightarrow} set.remove(1) \overset{\text{vis}}{\rightarrow} set.add(2)
```

```
set.read(): \{2\}
```
Write-conflict detection

- Operations updating the same object conflict, so cannot be causally independent:
  \[
  \forall e, f \in E. \ obj(e) = obj(f) \land update(op(e)) \land update(op(f)) \\
  \implies e \xrightarrow{vis} f \lor f \xrightarrow{vis} e
  \]

- Visibility totally orders transactions updating the same object \implies don't need replicated data types, don't need ar

- Can use sequential data types: from now on just sequential read-write registers
Transactional consistency zoo

Eventual consistency

Session guarantees

Causal consistency

Prefix consistency

Serializability

Parallel Snapshot Isolation

Snapshot Isolation
Transactional consistency zoo

Eventual consistency

Session guarantees

Causal consistency

Prefix consistency

Serializability

Causal consistency + write-conflict detection

Parallel Snapshot Isolation

Snapshot Isolation
Robustness
Application correctness

• Does an application satisfy a particular correctness property?

  Integrity invariants: account balance is non-negative

• Is an application robust against a particular consistency model?

  Application behaves the same as when using a strongly consistent database
Application correctness

• Does an application satisfy a particular correctness property?

  Integrity invariants: account balance is non-negative

• Is an application robust against a particular consistency model?

  Application behaves the same as when using a strongly consistent database
Parallel snapshot isolation

- Database with only sequential read-write registers

- Assume there is an implicit transaction writing initial values to all registers
PSI = the set of histories \((E, \text{so}, \sim)\) such that for some \(\text{vis}\):

- **No causal cycles:** \(\text{so} \cup \text{vis}\) is acyclic

- **Eventual visibility:** \(\forall e \in E. e \xrightarrow{\text{vis}} f\) for all but finitely many \(f \in E\)

- **Transaction indivisibility:**
  \[
  \forall e, f, e', f'. e \nolhd f \land e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f'
  \]

- **Causality preservation:** \((\text{so} \cup \text{vis})^+ \subseteq \text{vis}\)

- **Write-conflict detection:**
  \[
  \forall e, f \in E. \text{obj}(e) = \text{obj}(f) \land \text{op}(e) = \text{write}(-) \land \text{op}(f) = \text{write}(-) \\
  \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e
  \]

- **A read event returns the value written by the last preceding write in \(\text{vis}\)**
PSI = the set of histories \((E, \text{so}, \sim)\) such that for some vis:

- **No causal cycles**: \(\text{so} \cup \text{vis}\) is acyclic

- **Eventual visibility**: \(\forall e \in E. \ e \xrightarrow{\text{vis}} f\) for all but finitely many \(f \in E\)

- **Transaction indivisibility**:
  \[\forall e, f, e', f'. \ e \not\sim f \land e' \sim e \xrightarrow{\text{vis}} f \sim f' \implies e' \xrightarrow{\text{vis}} f\]

- **Causality preservation**: \((\text{so} \cup \text{vis})^+ \subseteq \text{vis}\)

- **Write-conflict detection**:
  \[\forall e, f \in E. \ \text{obj}(e) = \text{obj}(f) \land \text{op}(e) = \text{write}(\_) \land \text{op}(f) = \text{write}(\_)
  \implies e \xrightarrow{\text{vis}} f \lor f \xrightarrow{\text{vis}} e\]

- **A read event returns the value written by the last preceding write in vis**

---

Well-formed because of write-conflict detection
Dekker example

\[
\begin{align*}
x &.write(1) \\
&\text{vis} \leftrightarrow \text{so} \\
y &.read(): 0
\end{align*}
\]

\[
\begin{align*}
y &.write(1) \\
&\text{so} \leftrightarrow \text{vis} \\
x &.read(): 0
\end{align*}
\]
Dekker example

```
x.write(1)
y.read(): 0
```

```
y.write(1)
x.read(): 0
```

```
vis -> so
```

```
so -> vis
```

```
y.read(): 0
```

```
x.read(): 0
```
Dekker example

\[\text{x.write(1)}\quad \text{vis} \quad \text{so} \quad \text{y.read(): 0}\]

\[\text{y.write(1)}\quad \text{so} \quad \text{vis} \quad \text{x.read(): 0}\]
Transactional Dekker = write skew

x.write(1)
vis ⇄ so
y.read(): 0

y.write(1)
so ⇄ vis
x.read(): 0
Transactional Dekker = write skew

\[
\begin{align*}
&\text{x.write(1)} \\
&\text{vis} \Downarrow\text{so} \\
&\text{y.read(): 0}
\end{align*}
\]

\[
\begin{align*}
&\text{y.write(1)} \\
&\text{so} \Downarrow\text{vis} \\
&\text{x.read(): 0}
\end{align*}
\]

Not serializable, allowed by transactional causal consistency and parallel snapshot isolation
Transactional Dekker = write skew

\[
\begin{align*}
x &. \text{write}(1) \\
\text{vis} & \leftrightarrow \text{so} \\
y &. \text{read}(): 0
\end{align*}
\]

\[
\begin{align*}
y &. \text{write}(1) \\
\text{so} & \leftrightarrow \text{vis} \\
x &. \text{read}()(): 0
\end{align*}
\]
Transactional Dekker = write skew

```plaintext
x.write(1)
vis
y.read(): 0
so

y.write(1)
so
vis

x.read(): 0
```

Diagram showing two servers with data flow and transactional operations.
Independent reads of independent writes (IRIW)
Independent reads of independent writes (IRIW)

x.write(1)  y.write(1)  x.read: 1  y.read: 1

so

y.read: 0  x.read: 0

x written before y  y written before x
Independent reads of independent writes (IRIW)

Implementations: no causal dependency between the two writes

→ can be delivered in different orders at different replicas
Transactional IRIW = long fork

x.write(1)
y.write(1)
x.read: 1
y.read: 0
so
y.read: 1
x.read: 0
Transactional IRIW = long fork

x.write(1) → y.write(1) → x.read: 1

so

y.read: 0

so

y.read: 1

x.read: 0

Not serializable, allowed by transactional causal consistency and parallel snapshot isolation
Robustness

- Is an application robust against a particular consistency model?

  *Application behaves the same as when using a strongly consistent database*

Application behaves the same whether using a PSI or a serializable database: \([A]_{\text{PSI}} = [A]_{\text{SER}}\)
Robustness

- Application: set of transactional programs \{P_1, ..., P_n\}

```c
// tx lookup
int lookup() {
    return acct.bal;
}

// tx deposit
void deposit(int n) {
    acct.bal += n;
}
```

- Every program can generate multiple transactions at run time
- Simplification: every program is in its own session
Robustness

- Application: set of transactional programs \( \{P_1, ..., P_n\} \)

- Every program can generate multiple transactions at run time

- Simplification: every program is in its own session

- Checking robustness via static analysis: over-approximate the set of program behaviours
Application

$P_1 \quad P_2 \quad \ldots \quad P_n$
∀ PSI execution
∀ PSI execution

∃ serial execution
∀ PSI execution

∃ serial execution

Each read returns the value written by the last write
∀ PSI execution

∃ serial execution

Each read returns the value written by the last write
First determine if a given PSI execution is serializable

∀ PSI execution

∃ serial execution

Each read returns the value written by the last write
Build constraints on the serial order: relations on $E/\sim$ that should be included into $to/\sim$ - **transactional dependencies**

∀ PSI execution

∃ serial execution

Each read returns the value written by the last write
Write-read dependency ($wr$)

Given a PSI execution $(E, \sim, \text{vis})$ and $T, S \in E/\sim$.

$T \xrightarrow{wr} S \iff S$ reads a value written by $T$.
Write-read dependency (wr)

Given a PSI execution \((E, \sim, \text{vis})\) and \(T, S \in E/\sim\)

\[ T \xrightarrow{\text{wr}} x.\text{write}(\text{val}) \xrightarrow{\text{wr}} x.\text{read} : \text{val} \xrightarrow{\text{to/\sim}} S \]

\( T \xrightarrow{\text{wr}} S \iff S \text{ reads a value written by } T \)
Write-read dependency (wr)

Given a PSI execution \((E, \sim, \text{vis})\) and \(T, S \in E/\sim\)

\[ T \xrightarrow{\text{wr}} S \iff S \text{ reads a value written by } T \]

\[ T \xrightarrow{\text{wr}} S \iff T \neq S \land T \text{ contains the most recent write of an object } x \text{ visible to a read from } x \text{ in } S \text{ according to } \text{vis} \]
Write-write dependency (wr)

Given a PSI execution \((E, \sim, \text{vis})\) and \(T, S \in E/\sim\)

\[ T \xrightarrow{ww} x.\text{write}(old) \xrightarrow{ww} x.\text{write}(new) S \]

\(T \xrightarrow{ww} S \iff S\) overwrites a value written by \(T\)
Write-write dependency (wr)

Given a PSI execution \((E, \sim, \text{vis})\) and \(T, S \in E/\sim\)

\[
\begin{align*}
T \xrightarrow{\text{ww}} x.\text{write}(old) & \xrightarrow{\text{ww}} \text{to/\sim} \xrightarrow{\text{ww}} x.\text{write}(new) & S
\end{align*}
\]

\(T \rightarrow S \iff S\) overwrites a value written by \(T\)
Write-write dependency (wr)

Given a PSI execution \((E, \sim, \text{vis})\) and \(T, S \in E/\sim\)

\[
T \xrightarrow{\text{ww}} S \iff S \text{ overwrites a value written by } T
\]

\[
T \xrightarrow{\text{ww}} S \iff \text{T and S contain writes to the same object } x \text{ and } T \xrightarrow{\text{vis/\sim}} S
\]
Read-write dependency (rw)

T ⟷ x.read : old ➔ rw ➔ x.write(new) ⟷ S

rw
T ➔ S ⟷ T ≠ S ∧ S overwrites a value read by T

rw
T ➔ S ⟷ T ≠ S ∧ ∃Q. Q ➔ T ∧ Q ➔ S
Read-write dependency (rw)

T ⟷ S ⇔ T ≠ S ∧ S overwrites a value read by T

T ⟷ S ⇔ T ≠ S ∧ ∃Q. Q → T ∧ Q → S
Read-write dependency (rw)

\[ T \xrightarrow{rw} S \iff T \neq S \land S \text{ overwrites a value read by } T \]

\[ T \xrightarrow{rw} S \iff T \neq S \land \exists Q. Q \xrightarrow{wr} T \land Q \xrightarrow{ww} S \]
Read-write dependency (rw)

\[ T \xrightarrow{\text{rw}} S \iff T \neq S \land S \text{ overwrites a value read by } T \]

\[ T \xrightarrow{\text{rw}} S \iff T \neq S \land \exists Q. Q \xrightarrow{\text{wr}} T \land Q \xrightarrow{\text{ww}} S \]
Read-write dependency (rw)

T \xrightarrow{\text{rw}} S \iff T \neq S \land S \text{ overwrites a value read by } T

\begin{align*}
T \xrightarrow{\text{rw}} S & \iff T \neq S \land \exists Q. Q \xrightarrow{\text{rw}} T \land Q \xrightarrow{\text{ww}} S
\end{align*}
Read-write dependency (\(\text{rw}\))

\[ T \xrightarrow{\text{rw}} S \iff T \neq S \land S \text{ overwrites a value read by } T \]

\[ T \xrightarrow{\text{rw}} S \iff T \neq S \land \exists Q. Q \xrightarrow{\text{wr}} T \land Q \xrightarrow{\text{ww}} S \]
Read-write dependency (rw)

T \xrightarrow{\text{rw}} S \iff T \neq S \land S \text{ overwrites a value read by } T

T \xrightarrow{\text{rw}} S \iff T \neq S \land \exists Q. Q \xrightarrow{\text{wr}} T \land Q \xrightarrow{\text{ww}} S
Dependency graphs

• PSI execution \((E, \sim, \text{vis})\) \(\rightarrow\) dependency graph \((E/\sim, \text{wr}, \text{ww}, \text{rw})\)

• Theorem: If the dependency graph is acyclic, then the execution is serializable
If \((\text{wr} \cup \text{ww} \cup \text{wr})\) is acyclic, then there is a total order on \(E/\sim\) containing it [order-extension principle] \(\Rightarrow\) the desired order to
If \((wr \cup ww \cup wr)\) is acyclic, then there is a total order on \(E/\sim\) containing it [order-extension principle] \(\rightarrow\) the desired order to

Each read returns the value written by the last write in to?
\( T \xrightarrow{\text{wr}} S \iff T \neq S \land T \) contains the most recent write of an object \( x \) visible to a read from \( x \) in \( S \) according to \( \text{vis} \)

\( T \xrightarrow{\text{ww}} S \iff T \) and \( S \) contain writes to the same object \( x \) and \( T \xrightarrow{\text{vis/\sim}} S \)

\( T \xrightarrow{\text{rw}} S \iff T \neq S \land \exists Q. Q \xrightarrow{\text{wr}} T \land Q \xrightarrow{\text{ww}} S \)
If the dependency graph \((E/\sim, \text{wr}, \text{ww}, \text{rw})\) of a PSI execution \((E, \sim, \text{vis})\) is acyclic, then the execution is serializable.
If the dependency graph \((E/\sim, wr, ww, rw)\) of a PSI execution \((E, \sim, vis)\) is acyclic, then the execution is serializable

Transactional programs \(P_1, P_2, \ldots, P_n\)

\[\downarrow\]

Set of all their PSI executions \((E, \sim, vis)\)
If the dependency graph \((E/\sim, \text{wr}, \text{ww}, \text{rw})\) of a PSI execution \((E, \sim, \text{vis})\) is acyclic, then the execution is serializable.

\[
\begin{align*}
\text{Transactional programs } P_1, P_2, ..., P_n \\
\downarrow \\
\text{Set of all their PSI executions } (E, \sim, \text{vis}) \\
\downarrow \\
\text{Set of corresponding dependency graphs } (E/\sim, \text{wr}, \text{ww}, \text{rw})
\end{align*}
\]
If the dependency graph \((E/\sim, \text{wr}, \text{ww}, \text{rw})\) of a PSI execution \((E, \sim, \text{vis})\) is acyclic, then the execution is serializable.
If the dependency graph \((E/\sim, \text{wr}, \text{ww}, \text{rw})\) of a PSI execution \((E, \sim, \text{vis})\) is acyclic, then the execution is serializable.

Transaction programs \(P_1, P_2, ..., P_n\)

\[\downarrow\]

Set of all their PSI executions \((E, \sim, \text{vis})\)

\[\downarrow\]

Set of corresponding dependency graphs \((E/\sim, \text{wr}, \text{ww}, \text{rw})\)

\[\downarrow\]

Check \(\text{wr} \cup \text{ww} \cup \text{wr}\) is acyclic in each graph

Over-approximate the set of possible dependency graphs from the program text.
Static dependency graphs

- **Nodes**: transactional programs
- **Edges**: over-approximations of dependencies $\text{wr}^\#$, $\text{ww}^\#$, $\text{rw}^\#$

```plaintext
tx lookup() {
    return acct.bal
}

tx deposit(n) {
    acct.bal += n
}
```
Static dependency graphs

- Nodes: transactional programs
- Edges: over-approximations of dependencies $\text{wr}^\#$, $\text{ww}^\#$, $\text{rw}^\#$
- $T \xrightarrow{\text{wr}^\#} S \iff \exists x. \text{writes}(T, x) \land \text{reads}(T, x)$: over-approximated by static analyses (or even by hand)
Static dependency graphs

- Nodes: transactional programs
- Edges: over-approximations of dependencies \( \text{wr}^#, \text{ww}^#, \text{rw}^# \)
- \( T \xrightarrow{\text{wr}^#} S \iff \exists x. \text{writes}(T, x) \land \text{reads}(T, x): \text{over-approximated by static analyses (or even by hand)} \)
- Represents an over-approximation of all dynamic dependency graphs that can be produced by the programs

```plaintext
// tx lookup()
return acct.bal;

// tx deposit(n)
acct.bal += n;
```
Dynamic dependency graph $\rightarrow$ a subgraph of the static dependency graph

```
tx lookup() {
    return acct.bal
}
```

```
rx deposit(n) {
    acct.bal += n
}
```
Dynamic dependency graph \(\rightarrow\) a subgraph of the static dependency graph

```
tx lookup() {
  return acct.bal
}
```

```
tx deposit(n) {
  acct.bal += n
}
```
Dynamic dependency graph \(\rightarrow\) a subgraph of the static dependency graph

```plaintext
tx lookup() {
    return acct.bal
}
```

```plaintext
tx deposit(n) {
    acct.bal += n
}
```

Transactions arising from the same program map to the same node
Dynamic dependency graph ➔ a subgraph of the static dependency graph

\[
\text{tx lookup()} \{ \\
\quad \text{return acct.bal} \\
\}
\]

\[
\text{tx deposit(n)} \{ \\
\quad \text{acct.bal += n} \\
\}
\]

Edge in the dynamic graph ➔ corresponding edge in the static graph
Dynamic dependency graph \(\rightarrow\) a subgraph of the static dependency graph

```
Tx lookup() {
    return acct.bal
}
```

```
Tx deposit(n) {
    acct.bal += n
}
```

Edge in the dynamic graph \(\rightarrow\) corresponding edge in the static graph
Dynamic dependency graph ➔ a subgraph of the static dependency graph

```
// tx lookup()
return acct.bal

// tx deposit(n)
acct.bal += n
```

Edge in the dynamic graph ➔ corresponding edge in the static graph
Dynamic dependency graph ➔ a subgraph of the static dependency graph

tax lookup() {
    return acct.bal
}

tax deposit(n) {
    acct.bal += n
}

Edge in the dynamic graph ➔ corresponding edge in the static graph
Dynamic dependency graph → a subgraph of the static dependency graph

```
void tx lookup() {
    return acct.bal
}
```

```
void tx deposit(n) {
    acct.bal += n
}
```

Dynamic dependency graph

Cycle in the dynamic graph → cycle in the static graph
If the static graph is acyclic, so is the dynamic one
We're considering PSI executions: some cycles can't occur

```javascript
tx lookup() {
  return acct.bal
}
```

```javascript
tx deposit(n) {
  acct.bal += n
}
```

Dynamic dependency graph

We're considering PSI executions: some cycles can't occur

Cycle in the dynamic graph ➔ cycle in the static graph
If the static graph is acyclic, so is the dynamic one
T \xrightarrow{wr} S \iff T \neq S \land T \text{ contains the most recent write of an object } x \text{ visible to a read from } x \text{ in } S \text{ according to } \text{vis}

T \xrightarrow{ww} S \iff T \text{ and } S \text{ contain writes to the same object } x \text{ and } T \xrightarrow{vis/\sim} S
\[
\begin{align*}
\text{T} & \quad x.\text{write}(val) \quad \xrightarrow{\text{wr}} \quad x.\text{read} : val \quad \text{S} \\
\text{T} \quad \text{wr} \quad \text{S} & \iff \text{T} \neq \text{S} \land \text{T contains the most recent write of an object x visible to a read from x in S according to vis} \\
\text{T} & \quad x.\text{write}(old) \quad \xrightarrow{\text{ww}} \quad x.\text{write}(new) \quad \text{S} \\
\text{T} \quad \text{ww} \quad \text{S} & \iff \text{T and S contain writes to the same object x and T} \xrightarrow{\text{vis}} \text{S} \\
\text{wr} \cup \text{ww} & \subseteq \text{vis} ~/ \quad - \text{acyclic}
\end{align*}
\]
\[ T \xrightarrow{\text{wr}} S \iff T \neq S \land T \text{ contains the most recent write of an object } x \text{ visible to a read from } x \text{ in } S \text{ according to } \text{vis} \]

\[ T \xrightarrow{\text{ww}} S \iff T \text{ and } S \text{ contain writes to the same object } x \text{ and } T \xrightarrow{\text{vis/}} S \]

\[ \text{wr} \cup \text{ww} \subseteq \text{vis/} - \text{ acyclic} \]

PSI allows only cycles in \((\text{wr} \cup \text{ww} \cup \text{rw})\) with at least one \text{rw} edge
tx lookup() {
    return acct.bal
}

tax deposit(n) {
    acct.bal += n
}

Dynamic dependency graph ➔ a subgraph of the static dependency graph
tx lookup() {
    return acct.bal
}

tx deposit(n) {
    acct.bal += n
}

- Dynamic cycles with no rw edges aren't PSI ➔
don't represent robustness violations
tx lookup() {
    return acct.bal
}

tx deposit(n) {
    acct.bal += n
}

• Dynamic cycles with no rw edges aren't PSI ➜
  don't represent robustness violations

• Enough to check no cycles in (wr ∪ ww ∪ rw) with ≥ 1 rw
tx lookup() {
    return acct.bal
}

tx deposit(n) {
    acct.bal += n
}

• Dynamic cycles with no rw edges aren't PSI ➔
  don't represent robustness violations

• Enough to check no cycles in (wr ∪ ww ∪ rw) with ≥1 rw

• Enough to check no cycles in (wr# ∪ ww# ∪ rw#) with ≥1 rw#
Tightening up the criterion

PSI allows only cycles in \((\text{wr} \cup \text{ww} \cup \text{wr})\) with at least two distinct \text{rw} edges
Tightening up the criterion

PSI allows only cycles in \((\text{wr} \cup \text{ww} \cup \text{wr})\) with at least two distinct \text{rw} edges
Tightening up the criterion

PSI allows only cycles in \((wr \cup ww \cup wr)\) with at least two distinct \(rw\) edges
Tightening up the criterion

PSI allows only cycles in \((wr \cup ww \cup wr)\) with at least two distinct \(rw\) edges
Tightening up the criterion

PSI allows only cycles in \((\text{wr} \cup \text{ww} \cup \text{wr})\) with at least two distinct \(\text{rw}\) edges.

\[\downarrow\]

If \((\text{wr} \cup \text{ww} \cup \text{wr})\) for a PSI execution contains a cycle, then it also contains one:

- with at least two \(\text{rw}\) edges, and
- where all \(\text{rw}\) edges are due to distinct objects.
Transactional Dekker = write skew

x.write(0)
y.write(0)

x.write(1)
y.read(): 0

vis/~
y.write(1)
x.read(): 0
vis/~
Transactional Dekker = write skew

\[
x.\text{write}(0) \\
y.\text{write}(0)
\]

\[
\xrightarrow{\text{wr}(y)} \\
\xrightarrow{\text{wr}(x)}
\]

\[
x.\text{write}(1) \\
y.\text{read}(): 0
\]

\[
y.\text{write}(1) \\
x.\text{read}(): 0
\]
Transactional Dekker = write skew

x.write(0)
y.write(0)

vis/~
wr(y)
ww(x)

x.write(1)
y.read(): 0

vis/~
wr(x)
ww(y)

y.write(1)
x.read(): 0
Transactional Dekker = write skew

x.write(0)
y.write(0)

vis/~

wr(y)
ww(x)

x.write(1)
y.read(): 0

vis/~

wr(x)
ww(y)

rw(x)

y.write(1)
x.read(): 0
Transactional Dekker = write skew

x.write(0)
y.write(0)

x.write(1)
y.read(): 0

vis/~
wr(y)
ww(x)

rw(x)
y.write(1)
x.read(): 0

vis/~
wr(x)
ww(y)
Transactional Dekker = write skew

Cycle with 2 rw on different objects: allowed by PSI
Transactional IRIW = long fork

- `x.write(1)`
- `y.write(1)`
- `x.read : 1`
- `y.read : 0`
- `y.read : 1`
- `x.read : 0`
Transactional IRIW = long fork

\[ x.\text{write}(1) \xrightarrow{\text{vis/\sim, wr(x)}} x.\text{read} : 1 \]
\[ y.\text{write}(1) \xrightarrow{\text{vis/\sim, wr(y)}} y.\text{read} : 1 \]
Transactional IRIW = long fork

\[
x.\text{write}(1) \quad \xrightarrow{\text{vis/}\sim, \text{wr}(x)} \quad x.\text{read} : 1 \\
y.\text{write}(1) \quad \xrightarrow{\text{vis/}\sim, \text{wr}(y)} \quad y.\text{read} : 1 \\
x.\text{read} : 1 \quad \xrightarrow{\text{rw}(x)} \quad y.\text{read} : 0 \\
y.\text{read} : 0 \quad \xrightarrow{\text{vis/}\sim, \text{wr}(y)} \quad x.\text{read} : 0
\]
Transactional IRIW = long fork

x.write(1) → vis/~, wr(x) → x.read : 1
  rw(x) → y.read : 0

y.write(1) → rw(y) → y.read : 1
  vis/~, wr(y) → x.read : 0
Transactional IRIW = long fork

Cycle with 2 rw on different objects: allowed by PSI
Lost update anomaly

Not a valid PSI execution: violates write-conflict detection
Lost update anomaly

Not a valid PSI execution: violates write-conflict detection
Lost update anomaly

Not a valid PSI execution: violates write-conflict detection
Lost update anomaly

Not a valid PSI execution: violates write-conflict detection
Lost update anomaly

Not a valid PSI execution: violates write-conflict detection
**Lost update anomaly**

*Not a valid PSI execution: violates write-conflict detection*

*The 2 rw edges are due to the same object*
Static robustness criterion

If a dependency graph of a PSI execution contains a cycle, then it also contains one:

- with at least two \texttt{rw} edges, and
- where all \texttt{rw} edges are due to distinct objects

```plaintext
tx lookup() {
    return acct.bal
}

rw#(acct)
```

```plaintext
tx deposit(n) {
    acct.bal += n
}

wr#(acct)
```

```
ww#, rw#, wr#(acct)
```
Static robustness criterion

If a dependency graph of a PSI execution contains a cycle, then it also contains one:

- with at least two rw edges, and
- where all rw edges are due to distinct objects

No cycles in wr# ∪ ww# ∪ rw# with all rw# on different objects
Static robustness criterion

If a dependency graph of a PSI execution contains a cycle, then it also contains one:

- with at least two \( rw \) edges, and
- where all \( rw \) edges are due to distinct objects

No cycles in \( wr \cup ww \cup rw \) with all \( rw \) on different objects

\[\implies\] no such cycles in \( wr \cup ww \cup rw \)
Static robustness criterion

If a dependency graph of a PSI execution contains a cycle, then it also contains one:

- with at least two \texttt{rw} edges, and
- where all \texttt{rw} edges are due to distinct objects

\[ \text{tx lookup()} \{ \text{return acct.bal} \} \]
\[ \text{tx deposit(n)} \{ \text{acct.bal += n} \} \]

No cycles in \texttt{wr} \cup \texttt{ww} \cup \texttt{rw} with all \texttt{rw} on different objects

\[ \implies \text{no such cycles in } \texttt{wr} \cup \texttt{ww} \cup \texttt{rw} \]

\[ \implies \text{application is serializable} \]
Non-robustness

```plaintext
tx lookupAll() {
    return acct[0].bal
}

tx deposit(i, n) {
    acct[i].bal += n
}

lookupAll : 1/€100, 2/€0

lookupAll : 1/€0, 2/€100

deposit(1, €100)

vis/~

lookupAll : 1/€100, 2/€0

lookupAll : 1/€0, 2/€100

deposit(2, €100)

vis/~
```
Automatic robustness checking

- Methods for other consistency models are similar
- Basis for practical tools [Warszawski et al., SIGMOD'17, Brutschy et al., PLDI'18; Nagar et al., CONCUR'18]
- Static criterion on graphs sometimes used to prune the search space before a more expensive analysis with more semantic information
- Can be used for bug-finding in the absence of specifications
ABSTRACT
In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an ACIDRain attack, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and find many real-world ACIDRain vulnerabilities.
Implementing strong consistency
Designing consistency protocols

- So far implementations have been lightweight: "an operation can only be delivered after all its causal dependencies"

- In reality, designing consistency protocols and proving them correct is very difficult!

- Even more so for strong consistency protocols
Strong consistency

c.withdraw(100) : ?

c.withdraw(100) : ?
Strong consistency

Sombody has to order commands

c.withdraw(100) : ✔
c.withdraw(100) : ?
Strong consistency

Single server, clients send commands to the server
Strong consistency

Server totally orders commands and computes the sequence of results
Servers can crash! Need a fault-tolerant solution
Clients send commands to all replicas
Replicas may receive commands in different orders
A distributed protocol totally order commands: needs synchronisation.
State machine replication

Operations are deterministic \(\implies\) replicas compute the same sequence of results
State machine replication

 Implements sequential consistency (in fact, linearizability)
SMR requires solving a sequence of consensus instances: agree on the next command to execute.
Several nodes, which can crash

Each proposes a value
Consensus

- Several nodes, which can crash
- Each proposes a value
- All non-crashed nodes agree on a single value
• Challenge: asynchronous channels \implies can't tell a crashed node from a slow one!

• Assume only a minority of nodes can crash: a majority reach an agreement
The zoo of consensus protocols

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)
The zoo of consensus protocols

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)
The zoo of consensus protocols:

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)

Complex protocols: constant fight for better performance:

- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)
The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament’s protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [Computer-Communications Networks]: Distributed Systems—Network operating systems; D4.5 [Operating Systems]: Reliability—Fault-tolerance; J.1 [Administrative Data Processing]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the TOCS editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.
The Part-Time Parliament

LESLEI LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Leslie Lamport

Abstract

The Paxos algorithm, when presented in plain English, is very simple.
Paxos Made Simple

ROBBERT VAN RENESSE and DENIZ ALTINBUKEN, Cornell University

This article explains the full reconfigurable multidegree Paxos (or multi-Paxos) protocol. Paxos is by no means a simple protocol, even though it is based on relatively simple invariants. We provide pseudocode and explain it guided by invariants. We initially avoid optimizations that complicate comprehension. Next we discuss liveness, list various optimizations that make the protocol practical, and present variants of the protocol.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems—Network operating systems; D.4.5 [Operating Systems]: Reliability—Fault-tolerance

General Terms: Design, Reliability

Additional Key Words and Phrases: Replicated state machines, consensus, voting

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/2673577
Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov’s Viewstamped Replication [27, 20]), but it has several novel features:

- **Strong leader**: Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.

- **Leader election**: Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm.
In Search of an Understandable Consensus Algorithm

Abstract

Raft is a consensus algorithm for managing a file log. It produces a result equivalent to Paxos, but it is as efficient as Raft, and it is also more practical. The full explanation [15] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [16, 20, 21]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternate protocols.
Paxos Made Live - An Engineering Perspective
(2006 Invited Talk)

Tushar Chandra, Robert Griesemer, and Joshua Redstone

Google Inc.

ABSTRACT
We describe our experience in building a fault-tolerant database using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system.

Categories and Subject Descriptors
D.4.5 [Operating systems]: Reliability—Fault-tolerance; B.4.5 [Input/output and data communications]: Reliability, Testing, and Fault-Tolerance—Redundant design

General Terms
Experimentation, Performance, Reliability

Keywords
Experiences, Fault-tolerance, Implementation, Paxos database is just an example. As a result, the consensus problem has been studied extensively over the past two decades. There are several well-known consensus algorithms that operate within a multitude of settings and which tolerate a variety of failures. The Paxos consensus algorithm [8] has been discussed in the theoretical [16] and applied community [10, 11, 12] for over a decade.

We used the Paxos algorithm ("Paxos") as the base for a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudocode, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the lit-
Paxos Made Live - An Engineering Perspective
(2006 Invited Talk)

Tushar Chandra, Robert Griesemer, and Joshua Redstone

Google Inc.

ABSTRACT
We describe our experience in building a fault-tolerant system using the Paxos consensus algorithm. Despite the existing literature in the field, building such a data structure proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we made for them. Our measurements indicate that we are building a competitive system.

Categories and Subject Descriptors
D.4.5 [Operating systems]: Reliability—Fault-tolerance; B.4.5 [Input/output and data communications]: Reliability, Testing, and Fault-Tolerance—Redundant design

General Terms
Experimentation, Performance, Reliability

Keywords
Experiences, Fault-tolerance, Implementation, Paxos

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.

- While Paxos can be described with a page of pseudocode, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the lit-
Paxos Made Live
(2008)

Tushar Chandra, Rod

ABSTRACT
We describe our experience in building a fault-tolerant database using the Paxos consensus algorithm. Despite existing literature in the field, building such a database to be non-trivial. We describe selected algorithm engineering problems encountered, and the solutions we devised for them. Our measurements indicate that we have a competitive system.

Categories and Subject Descriptors
D.4.5 [Operating systems]: Reliability—Fault-tolerance
B.4.5 [Input/output and data communication]: Reliability, Testing, and Fault-Tolerance—Redundant and replicated systems

General Terms
Experimentation, Performance, Reliability

Keywords
Experiences, Fault-tolerance, Implementation, Paxos

5.1 Handling disk corruption
Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica’s disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file. The latter may be indistinguishable from a new replica with an empty disk—we detect this case by having a new replica leave a marker in GFS after start-up. If this replica ever starts again with an empty disk, it will discover the GFS marker and indicate that it has a corrupted disk.

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

production-ready system involved implementing many features and optimizations—some published in the lit-
5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica’s disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file. The latter may be indistinguishable from a new replica with an empty disk — we detect this case by having a new replica leave a marker in GFS that it has a corrupted disk.

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.
Another application: blockchain

• Blockchain = using consensus to agree on a sequence of blocks in a ledger

• Tolerates malicious behaviour: some nodes may deviate from the protocol

• Many protocols descended from Paxos
Facebook’s Libra pitches to be the future of money

Rory Cellan-Jones
Technology correspondent
@BBCRoryCJ

18 June 2019

It is a hugely ambitious - some might say megalomaniacal - project to create a new global currency. Facebook's David Marcus tells me it is about giving billions of people more freedom with money and "righting the many wrongs of the present system".

The message is this is not some little side project a small team at the Facebook's Mark Zuckerberg will throw a few resources behind and forget.

However, Libra faces huge regulatory hurdles. It will have to pass muster with Basel II, a new Victorian-era rulebook that is supposed to prevent another financial crisis. This shines a spotlight on over 300 global financial institutions who have signed up to the project.
[PODC'19]

HotStuff: BFT Consensus with Linearity and Responsiveness

Maofan Yin
Cornell University
VMware Research

Dahlia Malkhi
VMware Research

Michael K. Reiter
UNC-Chapel Hill
VMware Research

Guy Golan Gueta
VMware Research

Ittai Abraham
VMware Research

ABSTRACT

We present HotStuff, a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model. Once network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of actual (vs. maximum) network delay—a property called *responsiveness*—and with communication complexity that is linear in the number of replicas. To our knowledge, HotStuff is the first partially synchronous BFT replication protocol exhibiting these combined properties. Its simplicity enables it to be further pipelined and simplified into a practical, concise protocol for building large-scale replication services.

CCS CONCEPTS

• Software and its engineering → Software fault tolerance;
• Security and privacy → Distributed systems security.

KEYWORDS

Byzantine fault tolerance; consensus; responsiveness; scalability; blockchain

stabilization time (GST). In this model, $n \geq 3f + 1$ is required for non-faulty replicas to agree on the same commands in the same order (e.g., [12]) and progress can be ensured deterministically only after GST [27].

When BFT SMR protocols were originally conceived, a typical target system size was $n = 4$ or $n = 7$, deployed on a local-area network. However, the renewed interest in Byzantine fault-tolerance brought about by its application to blockchains now demands solutions that can scale to much larger $n$. In contrast to permissionless blockchains such as the one that supports Bitcoin, for example, so-called permissioned blockchains involve a fixed set of replicas that collectively maintain an ordered ledger of commands or, in other words, that support SMR. Despite their permissioned nature, numbers of replicas in the hundreds or even thousands are envisioned (e.g., [30, 42]). Additionally, their deployment to wide-area networks requires setting $\Delta$ to accommodate higher variability in communication delays.

The scaling challenge. Since the introduction of PBFT [20], the first practical BFT replication solution in the partial synchrony model, numerous BFT solutions were built around its core two
PODC'19

HotStuff: BFT Consensus with Linearity and Responsiveness

Maofan Yin
Cornell University
VMware Research

Dahlia Malkhi
VMware Research

Michael K. Reiter
UNC-Chapel Hill
VMware Research

Guy Golan Gueta
VMware Research

Ittai Abraham
VMware Research

ABSTRACT
We present HotStuff, a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model. Once network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of actual (vs. maximum) network delay—a property called responsiveness—and with communication complexity that is linear in the number of replicas. To our knowledge, HotStuff is the first partially synchronous BFT replication protocol exhibiting its simplicity enables it to be further piloted for practical, concise protocol for building services.

CCS CONCEPTS
- Software and its engineering → Security and privacy → Distributed systems

KEYWORDS
Byzantine fault tolerance; consensus; responsiveness; scalability; blockchain

stabilization time (GST). In this model, $n \geq 3f + 1$ is required for non-faulty replicas to agree on the same commands in the same order (e.g., [12]) and progress can be ensured deterministically only after GST [27].

When BFT SMR protocols were originally conceived, a typical target system size was $n = 4$ or $n = 7$, deployed on a local-area network. However, the renewed interest in Byzantine fault-tolerance brought about by its application to blockchains now demands solutions that can scale to much larger clusters. In contrast to previous implementations, HotStuff improves on existing performance by a factor of $O(\log n)$ in the partially synchronous regime; it achieves this by employing a novel leader election mechanism.

ACKNOWLEDGMENTS
We are thankful to Mathieu Baudet, Avery Ching, George Danezis, François Garillot, Zekun Li, Ben Maurer, Kartik Nayak, Dmitri Perelman, and Ling Ren, for many deep discussions of HotStuff, and to Mathieu Baudet for exposing a subtle error in a previous version posted to the ArXiv of this manuscript.
• 2f+1 nodes, at most f can crash
• Each node proposes a value
• All non-crashed nodes agree on a single value
• **Acceptors** = members of parliament: can vote to accept a value, majority (quorum) wins
• **Acceptors** = members of parliament: can vote to accept a value, majority *(quorum)* wins
• **Leader** = parliament speaker: proposes its value to vote on
• Good for state-machine replication: can elect the leader once and get it to process multiple commands
• **Phase 1:** a prospective leader convinces a quorum of acceptors to accept its authority
• Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority
• **Phase 1:** a prospective leader convinces a quorum of acceptors to accept its authority
• **Phase 1**: a prospective leader convinces a quorum of acceptors to accept its authority
• **Phase 1**: a prospective leader convinces a quorum of acceptors to accept its authority

• **Phase 2**: the leader gets a quorum of acceptors to accept its value and replies to the client
• **Phase 1:** a prospective leader convinces a quorum of acceptors to accept its authority

• **Phase 2:** the leader gets a quorum of acceptors to accept its value and replies to the client
• **Phase 1:** a prospective leader convinces a quorum of acceptors to accept its authority

• **Phase 2:** the leader gets a quorum of acceptors to accept its value and replies to the client
• **Phase 1:** a prospective leader convinces a quorum of acceptors to accept its authority

• **Phase 2:** the leader gets a quorum of acceptors to accept its value and replies to the client
• **Phase 1:** a prospective leader convinces a quorum of acceptors to accept its authority

• **Phase 2:** the leader gets a quorum of acceptors to accept its value and replies to the client
Phase 1: a prospective leader convinces a quorum of acceptors to accept its authority.

Phase 2: the leader gets a quorum of acceptors to accept its value and replies to the client.
• Problem: node 3 may wake up, form a quorum of 1 and 3, and accept value $v_3$. 
• Problem: node 3 may wake up, form a quorum of 1 and 3, and accept value $v_3$

• Need to ensure once a value is chosen by a quorum, it can’t be changed

• Use ballot numbers to distinguish different votes: unique for each potential leader
• **Phase 1**: a prospective leader chooses a ballot b and convinces a quorum of acceptors to switch to b

• **Acceptors** switches only if it’s current ballot is smaller
• **Phase 1:** a prospective leader chooses a ballot b and convinces a quorum of acceptors to switch to b

• **Acceptor switches** only if it’s current ballot is smaller
• **Phase 1:** a prospective leader chooses a ballot b and convinces a quorum of acceptors to switch to b

• Acceptor switches only if it’s current ballot is smaller
• **Phase 2**: the leader sends its value tagged with its ballot number

• **Acceptor** only accepts a value tagged with the ballot it is in
Phase 2: the leader sends its value tagged with its ballot number

Acceptor only accepts a value tagged with the ballot it is in
• **Phase 2**: the leader sends its value tagged with its ballot number

• Acceptor only accepts a value tagged with the ballot it is in
Leader#: 2
Ballot#: b
Accepted: v₂@b

Leader#: 2 ✔
Ballot#: b
Accepted: v₂@b ✔
Reply v₂ to client

Leader#: ?
Ballot#: 0
Accepted: ?
<table>
<thead>
<tr>
<th>Leader#</th>
<th>Ballot#</th>
<th>Accepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b</td>
<td>v₂@b</td>
</tr>
<tr>
<td>✔</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Leader#</th>
<th>Ballot#</th>
<th>Accepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b</td>
<td>v₂@b ✔</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reply v₂ to client</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Leader#</th>
<th>Ballot#</th>
<th>Accepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>

- Need to ensure once a value is chosen by a quorum, it can’t be changed
- Need do change Phase 1 to restrict which values can be proposed
1
Leader#: 2
Ballot#: b
Accepted: v₂@b

2
Leader#: 2 ✔️
Ballot#: b
Accepted: v₂@b✔️
Reply v₂ to client

3
Leader#: 3
Ballot#: b’ > b
Accepted: ?

b’
Phase 1: acceptor sends to the prospective leader its value and the ballot it was accepted at.

If some acceptor has accepted a value, the leader proposes the value accepted at the highest ballot number.
• **Phase 1**: acceptor sends to the prospective leader its value and the ballot it was accepted at

• If some acceptor has accepted a value, the leader proposes the value accepted at the highest ballot number
• Phase 1: acceptor sends to the prospective leader its value and the ballot it was accepted at

• If some acceptor has accepted a value, the leader proposes the value accepted at the highest ballot number

• Ensures the value chosen will not be changed \(\implies\) nodes don't disagree about the chosen value
Key invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot b' > b will also propose v

- Ensures the value chosen will not be changed ⟹ nodes don't disagree about the chosen value
Invariant: If a quorum $Q$ accepted a value $v$ at ballot $b$, then any leader of a ballot $b' > b$ may only propose $v$
Proof of the key invariant

- Invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot \( b' > b \) may only propose v.

- Fix an execution of a protocol and assume that in this execution Q accepted \( v@b \).
Proof of the key invariant

- Invariant: If a quorum Q accepted a value v at ballot b, then any leader of a ballot $b' > b$ may only propose v.

- Fix an execution of a protocol and assume that in this execution Q accepted $v@b$.

- We prove by induction on $b'$ that: for any $b' > b$, $\text{leader}(b')$ may only propose v.
Proof of the key invariant

- Invariant: If a quorum Q accepted a value \( v \) at ballot \( b \), then any leader of a ballot \( b' > b \) may only propose \( v \).

- Fix an execution of a protocol and assume that in this execution Q accepted \( v \)@\( b \).

- We prove by induction on \( b' \) that: for any \( b' > b \), \( \text{leader}(b') \) may only propose \( v \).

- Consider \( b' > b \) and assume \( \text{leader}(b'') \) may only propose \( v \) if \( b < b'' < b' \). We prove that \( \text{leader}(b') \) may only propose \( v \).
• $Q$ accepted $v@b$
• $b' > b$
• $\text{leader}(b'')$ may only propose $v$ if $b < b'' < b'$
• Q accepted $v_b$
• $b' > b$
• $\text{leader}(b'')$ may only propose $v$ if $b < b'' < b'$

• $\text{leader}(b')$ gets support from a quorum $Q'$ before proposing
- $Q$ accepted $v@b$
- $b' > b$
- $\text{leader}(b'')$ may only propose $v$ if $b < b'' < b'$

- $\text{leader}(b')$ gets support from a quorum $Q'$ before proposing

- $Q \cap Q' \neq \emptyset \implies \exists$ process $p \in Q \cap Q'$ which both voted for $\text{leader}(b')$ and accepted $v@b$
- $Q$ accepted $v@b$
- $b' > b$
- $\text{leader}(b'')$ may only propose $v$ if $b < b'' < b'$

- $\text{leader}(b')$ gets support from a quorum $Q'$ before proposing
- $Q \cap Q' \neq \emptyset \implies \exists \text{ process } p \in Q \cap Q'$ which both voted for $\text{leader}(b')$ and accepted $v@b$
- $p$ couldn't accept $v@b$ after voting for $\text{leader}(b')$: after voting, $p$ joins $b'$ and rejects all messages with ballot $b < b'$
• Q accepted v@b
• b’ > b
• leader(b’’) may only propose v if b < b’’ < b’

• leader(b’) gets support from a quorum Q’ before proposing
• Q \cap Q’ \neq \emptyset \implies \exists \text{ process } p \in Q \cap Q’ \text{ which both voted for leader(b’)} and accepted v@b
• p couldn't accept v@b after voting for leader(b’): after voting, p joins b’ and rejects all messages with ballot b < b’
• p accepted v@b before voting for leader(b’)

Q accepted v@b
b’ > b
leader(b’’) may only propose v if b < b’’ < b’
• Q accepted v@b
• b’ > b
• leader(b’’) may only propose v if b < b’’ < b’

• leader(b’) gets support from a quorum Q’ before proposing

• Q ∩ Q’ ≠ ∅ ⟹ ∃ process p ∈ Q ∩ Q’ which both voted for leader(b’)
  and accepted v@b

• p couldn't accept v@b after voting for leader(b’): after voting, p joins b’
  and rejects all messages with ballot b < b’

• p accepted v@b before voting for leader(b’)

• p’s ballot when voting for leader(b’) is b_p ≥ b > 0, and it will reply with
  v’@b_p for some value v’
• Q accepted v@b
• b' > b
• leader(b'') may only propose v if b < b'' < b'

• leader(b') gets support from a quorum Q' before proposing

• Q ∩ Q' ≠ ∅ ⟺ ∃ process p ∈ Q ∩ Q' which both voted for leader(b') and accepted v@b

• p couldn't accept v@b after voting for leader(b'): after voting, p joins b' and rejects all messages with ballot b < b'

• p accepted v@b before voting for leader(b')

• p's ballot when voting for leader(b') is b_p ≥ b > 0, and it will reply with v'@b_p for some value v'

• leader(b') can't propose its own value, has to pick one accepted at the highest ballot b_max ≥ b in the votes it got
• Q accepted \( v\@b \)
• \( b' > b \)
• \( \text{leader}(b'') \) may only propose \( v \) if \( b < b'' < b' \)
• \( b_{\text{max}} \geq b \)

\[b_{\text{max}} = b:\]
• Q accepted $v@b$
• $b' > b$
• $\text{leader}(b'')$ may only propose $v$ if $b < b'' < b'$
• $b_{\max} \geq b$

$$b_{\max} = b:$$

• A leader makes a single proposal per ballot, and $Q$ accepted $v@b \implies$ any vote $v'@b_{\max}$ for $\text{leader}(b')$ must have $v' = v$
• Q accepted $v@b$
• $b' > b$
• leader($b''$) may only propose $v$ if $b < b'' < b'$
• $b_{\text{max}} \geq b$

$b_{\text{max}} = b$: 

• A leader makes a single proposal per ballot, and Q accepted $v@b \implies$ any vote $v'@b_{\text{max}}$ for leader($b'$) must have $v' = v$
• leader($b'$) has to choose $v$, QED.
• Q accepted $v@b$
• $b' > b$
• $\text{leader}(b'')$ may only propose $v$ if $b < b'' < b'$
• $b_{\text{max}} \geq b$

$b_{\text{max}} > b$:
• Q accepted \( v@b \)
• \( b' > b \)
• \( \text{leader}(b'') \) may only propose \( v \) if \( b < b'' < b' \)
• \( b_{\text{max}} \geq b \)

\( b_{\text{max}} > b: \)

• \( b_{\text{max}} < b' \), since processes only vote for leaders of higher ballots
• Q accepted v@b
• b' > b
• \( \text{leader}(b''') \) may only propose v if \( b < b''' < b' \)
• \( b_{\text{max}} \geq b \)

\( b_{\text{max}} > b: \)

• \( b_{\text{max}} < b' \), since processes only vote for leaders of higher ballots
• By induction hypothesis leader(\( b_{\text{max}} \)) could only propose v
• Q accepted v@b
• b' > b
• leader(b'') may only propose v if b < b'' < b'
• b_{max} \geq b

\textbf{b_{max} > b:}

• b_{max} < b', since processes only vote for leaders of higher ballots
• By induction hypothesis leader(b_{max}) could only propose v
• Processes that accepted a value at b_{max} could only accept v
• Q accepted v@b
• b’ > b
• leader(b’’) may only propose v if b < b’’ < b’
• b_max ≥ b

b_max > b:

• b_max < b’, since processes only vote for leaders of higher ballots
• By induction hypothesis leader(b_max) could only propose v
• Processes that accepted a value at b_max could only accept v
• Any vote v’@b_max for leader(b’) must have v’ = v
• $Q$ accepted $v@b$
• $b' > b$
• $\text{leader}(b'')$ may only propose $v$ if $b < b'' < b'$
• $b_{\max} \geq b$

$b_{\max} > b$:

• $b_{\max} < b'$, since processes only vote for leaders of higher ballots
• By induction hypothesis $\text{leader}(b_{\max})$ could only propose $v$
• Processes that accepted a value at $b_{\max}$ could only accept $v$
• Any vote $v'@b_{\max}$ for $\text{leader}(b')$ must have $v' = v$
• $\text{leader}(b')$ has to choose $v$, QED.
**Key invariant:** If a quorum Q accepted a value v at ballot b, then any leader of a ballot $b' > b$ will also propose v

Ensures nodes don't disagree about the chosen value
Multi-Paxos

State machine replication requires solving a sequence of consensus instances
Multi-Paxos

State machine replication requires solving a sequence of consensus instances

- **Naive solution:** execute a separate Paxos instance for each sequence element
- **Multi-Paxos:** execute Phase 1 once for multiple sequence elements
Paxos verification

- Lots of work on formally verifying Paxos-like protocols in theorem provers or semi-automatic systems

- Fully automatic verification is an open problem
The end

• Spectrum of data consistency models in distributed systems

• Downsides of weakening consistency can be mitigated by verification techniques and programming abstractions: replicated data types, transactions

• Proving correctness of consistency protocols is a verification challenge