Eventually Consistent Transactions

Sebastian Burckhardt[1], Daan Leijen[1],
Manuel Fähndrich[1], and Mooley Sagiv[2]

presented by Dominic Meier

4/17/2013

[1] Microsoft Research
[2] Tel-Aviv University

Goal

- Eventual consistency in a distributed system is nice. It allows temporarily disconnected replicas to **remain fully available**.

- Find answers to the questions:

  1. How to provide consistency guarantees that are **as strong as possible** without losing **lazy consensus**?

  2. How to effectively **understand and implement** systems that provide these guarantees?

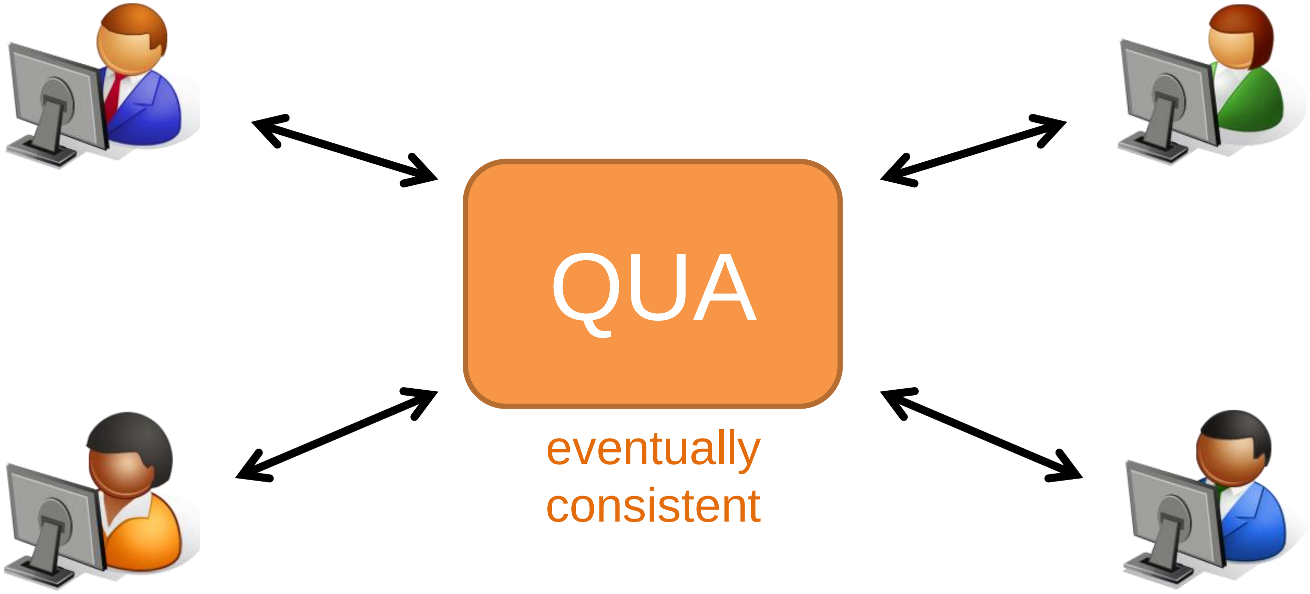
Outline

- Model Assumptions
- Important Definitions
- Sequential vs. Eventual Consistency
- Revision Consistency **HOT**
    - Construction and Properties of **Revision Diagrams**
    - **Theorem**: Revision Consistency ⇨ Eventual Consistency
- Conclusion and Future Work

Model Assumptions

- Distributed system.
- Multiple participants (*clients*).
- One logical database, referred to as a *Query-Update-Automaton (QUA)*.
- Clients issue **eventually consistent transactions**, that cannot fail and never roll back.
- All code runs inside transactions.

Model Assumptions



QUA

eventually consistent

- This is a **logical view** of the situation.
- The state of the *QUA* might be **distributed** and **temporarily inconsistent**.

Query-Update Interface & Automaton

Definition

A *query-update* *interface* is a tuple (Q, V, U) where

- Q is a set of **query operations**.
- V is a set of **values** returned by queries.
- U is a set of **update operations**.

Definition

A *query-update* *automaton* *(QUA)* for interface (Q, V, U) is a tuple (S, s_0) together with an **interpretation**, where

- S is a set of states
- s_0 is the initial state
- $q^\#$ is an **interpretation of query** $q \in Q$ as a function $S \to V$
- $u^\#$ is an **interpretation of command** $u \in Q$ as a function $S \to S$

Histories

Definition

A history H is a **map** that maps each client $c \in C$ to a finite or infinite **sequence** $H(c)$ of following events:

- $u \in U$ is an **update** issued by the client.
- (q, v) represents a **query** with its **return value**.
- yield **commits** transactions.

Example:

CLIENT #1

```
x := load(a);
store(b, x);
yield;
y := load(b);
store(a, y);
yield;
```

CLIENT #2

```
store(c, 5);
yield;
i := load(c);
store(c, i+2);
yield;
```

Consistency Models

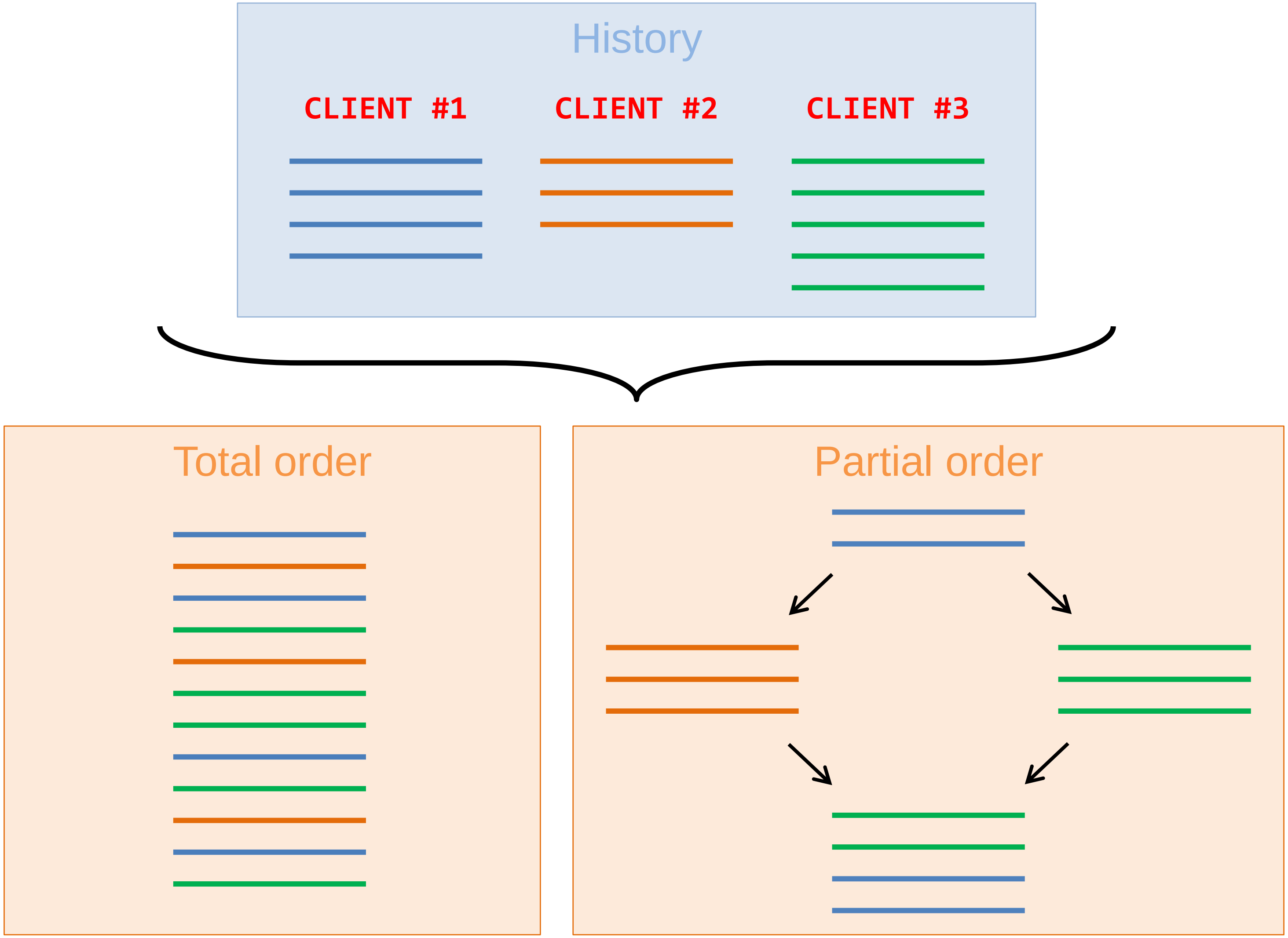
- Sequential Consistency:

  “The result of any execution is the same as if the operations of all the processors were executed in some sequential order, while retaining the program order.”

- Eventual Consistency:

  “If no new updates are made to a data object, eventually all accesses will return a consistent value.”

Enhance History with Additional Orders



History

CLIENT #1 **CLIENT #2** **CLIENT #3**

Total order

Partial order

Sequentially Consistent History

- Find a single partial order $<$ over all events in a given *history*, with the following properties:
  - Compatibility with **program order**.
  - Past events are **totally ordered**.
  - Transactions are **atomic**.
  - Transactions are executed in **isolation**.
  - Committed transactions are **eventually delivered** to all participants.

Sequential consistency **does not** tolerate temporary network partitions!

Eventually Consistent History

- Instead of one partial order, we try to find two:
  - Visibility order $<_v$
  - Arbitration order $<_a$

- The visibility order defines which events' **effects** are visible to which other events.

- The arbitration order defines the **relative order** of past events.

Eventual consistency tolerates temporary network partitions!

Eventual Consistency in Related Work

- In order to arbitrate events, two common approaches exist:
    - Use **timestamps**, actual or logical.
    - Make updates **commutative**.

- This paper suggests a different approach, which does not require any of the above.
    - Main contribution of the paper.

Write Stabilization Problem

Alice	Bob	Robinson
`update();` **`yield;`**	`update();` **`yield;`**	`update();` **`yield;`**

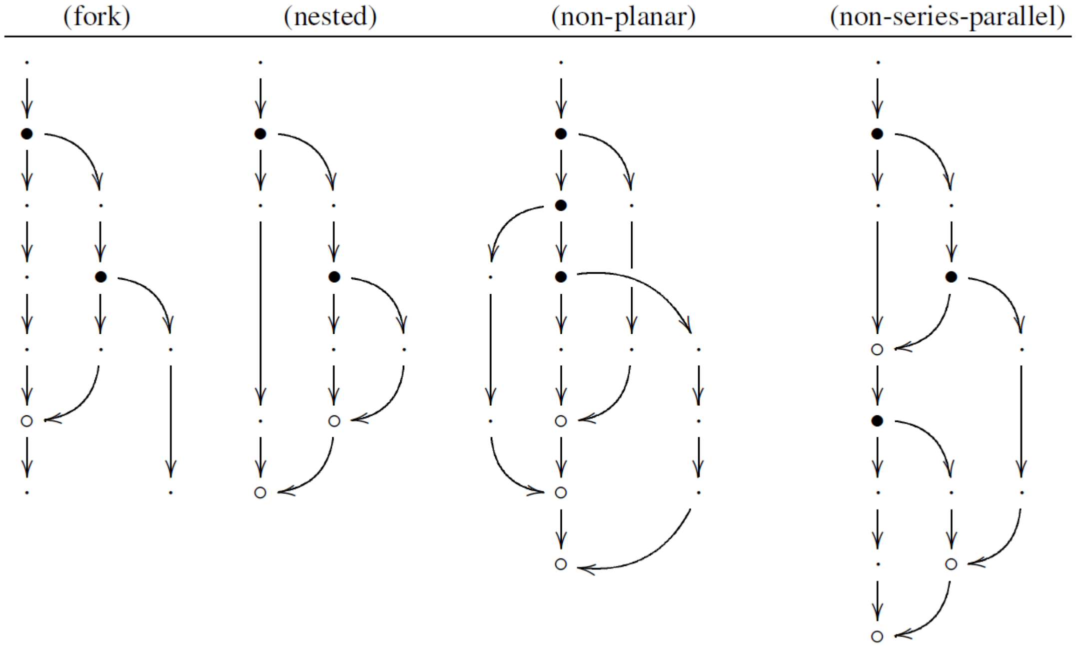
`Repeat 1000x:`
`    update();`
`    yield;`
Cannot stabilize!
	`Repeat 1000x:`
	`    update();`
	`    yield;`
	Cannot stabilize!
		Robinson disconnects!
		

Perform important update!

Solution:
Simply order Robinson's update **after** all the others!

Revision Diagrams

- A definition of Eventual Consistency does not **by itself** give **guidelines** as to how to build a system that is eventually consistent.
- That’s why *Revision Diagrams* are introduced.
- Examples:

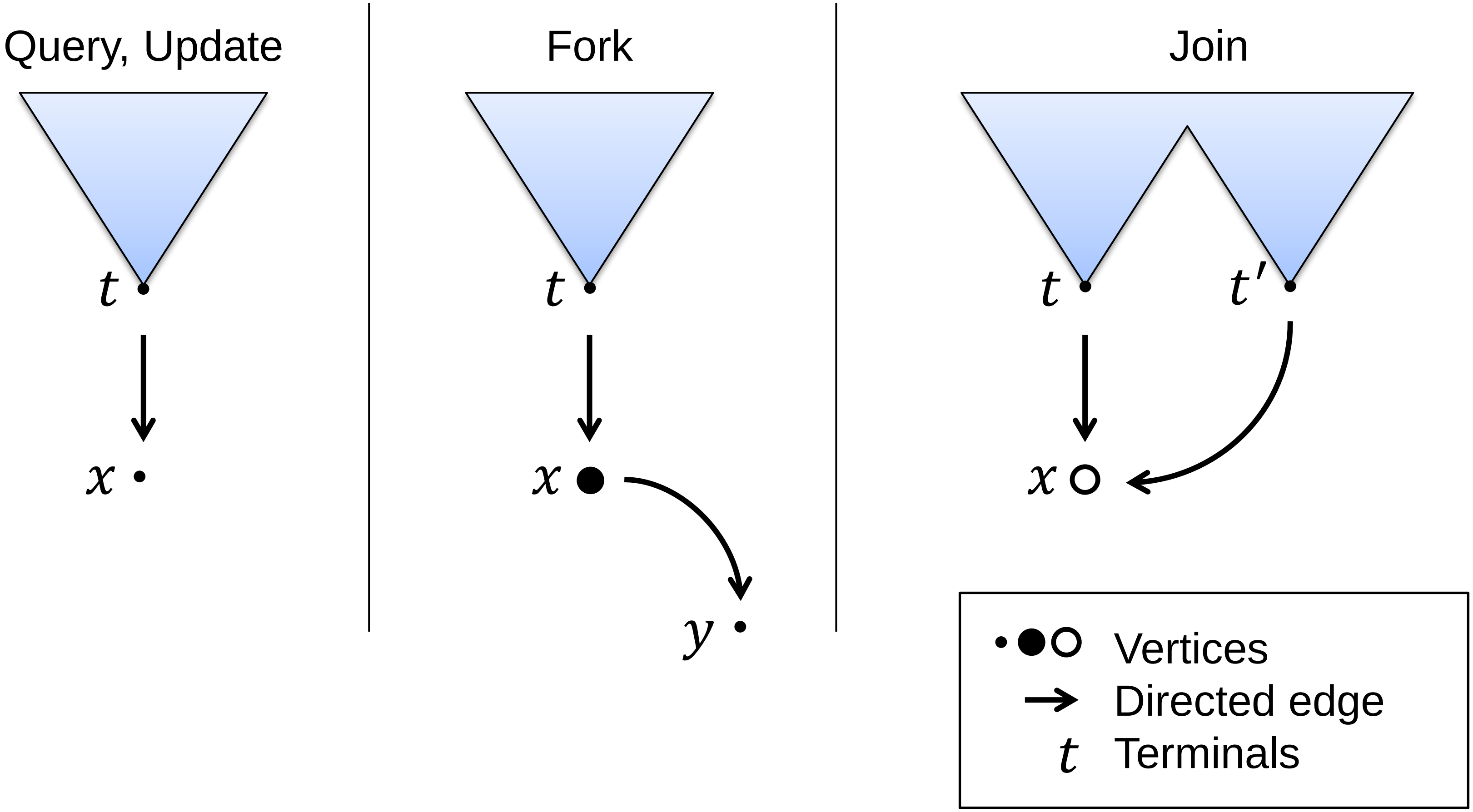


Revisions

- Revisions are **logical replicas of the state**.
- Clients work with **one revision at a time**, and can perform operations on it.
- **Reconciliation** happens during a so-called *join* operation between two revisions.

Revision Diagram: Construction Rules

- Start with **root vertex** as the only terminal, and then:



Join Condition

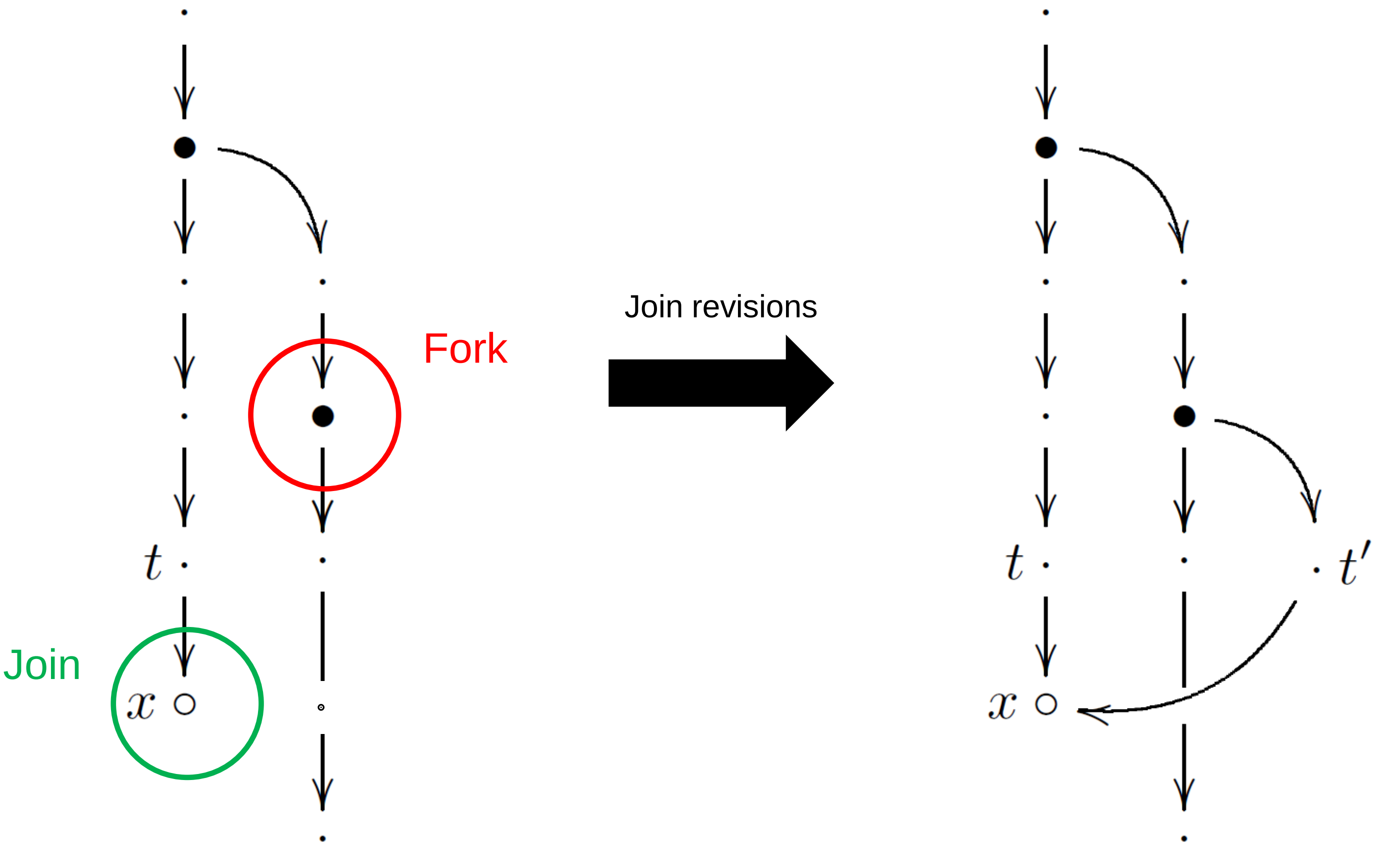
- In order for revision diagrams to be eventually consistent, the **join condition** needs to be satisfied.

Definition

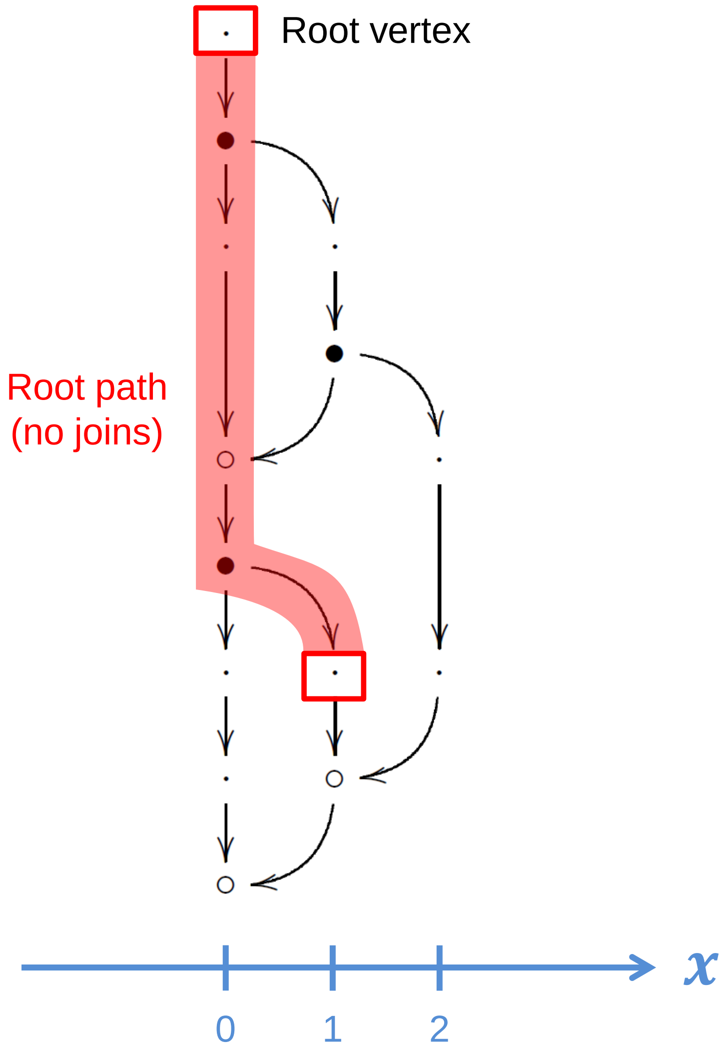
Join condition: The vertex that forked the joined revision **must reach** the *join* vertex

- This establishes important validity conditions and is needed for the proof of the upcoming theorem.

Example of non-satisfied Join Condition



Graph Properties of Revision Diagrams



- Vertices of the same revision have the **same x-coordinate**.
- *Fork* vertices spawn **new revisions to the right**.
- *Join* vertices **merge revisions coming from the right**.

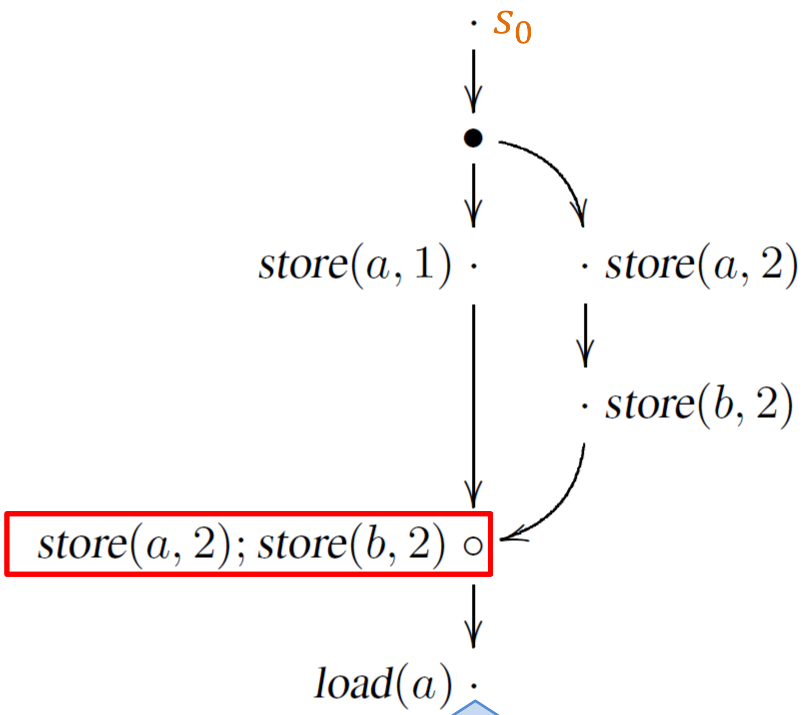
Effects of the Vertices

Definition

For any vertex x, we let the **effect** of x be a function $x^{\circ}: S \to S$ defined inductively as follows:

- If x is a *start*, *fork* or *query* vertex, there is no effect.
- If x is an **update** vertex for some update operation, then the effect is that update.
- If x is a *join* vertex, then the effect is the **composition of all effects** in the joined revision.

Update Effects



s_0

$store(a,1)$

$store(a,2)$

$store(b,2)$

$store(a,2); store(b,2)$

$load(a)$

$$load(a)^{\#}(store(b,2)^{\#}\left(store(a,2)^{\#}\left(store(a,1)^{\#}(s_0)\right)\right)) = 2$$

Revision Diagrams ⇔ Histories

History

CLIENT #1

```
x := load(a);
store(b, x);
yield;
y := load(b);
store(a, y);
yield;
```

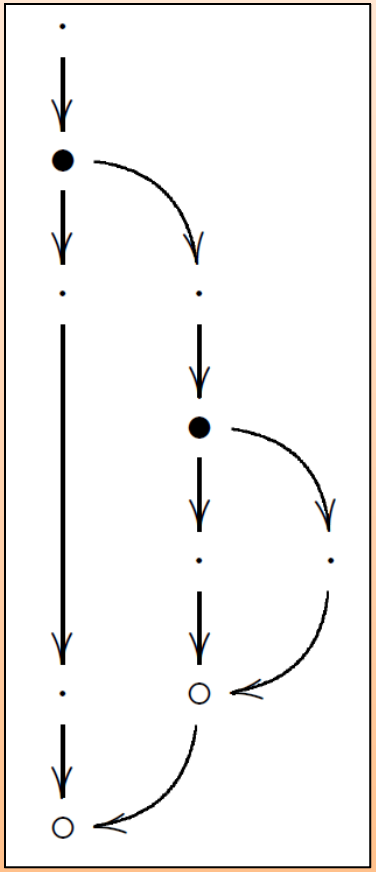
CLIENT #2

```
store(c, 5);
yield;
i := load(c);
store(c, i+2);
yield;
```

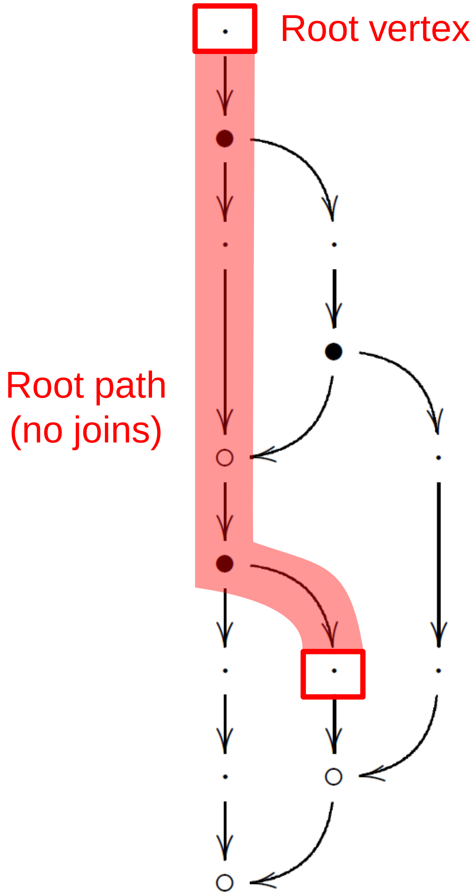
revision-consistent

Witness for

Revision Diagram



Requirements of Witness Diagrams



- Query events match the **path-result**.
- Successive non-yield operations of the same client are connected by a **vertical edge**.
- The beginning of a transaction **must be reachable** from the end of the previous transaction of the same client.

Neglected Vertices

Definition

A vertex x is **neglected**, if there exists an **infinite number** of vertices y such that there is no path from x to y.

- This would mean that the operation associated to this vertex could **starve**, i.e. could **not eventually be delivered**.

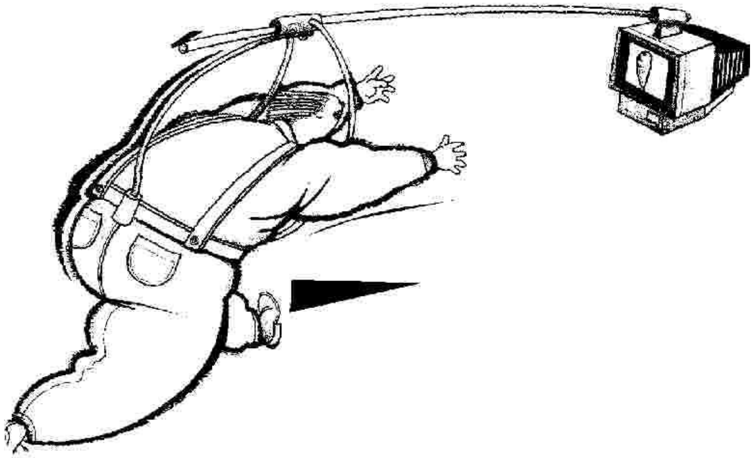


Image: © http://www.alternativedamm.de/

Theorem

Theorem

Let H be a history. If there exists a **witness diagram** for H such that no committed events are neglected, then H is **eventually consistent**.

- In the paper, a **proof** is included.
- Note that the **converse is not true**, i.e. if H is eventually consistent, there **might not** exist a witness diagram for it.

System Implementation

- Two **eventually consistent systems** might look like:

Single Synchronous Server Model

- Single server
- Multiple clients
- The server can spawn clients
- Clients can join the server
- Transactions are **committed** by clients by joining and forking again.

Pro:	Simple and intuitive
Contra:	Clients **block** if they have no connection

Server Pool Model

- Multiple servers
- Multiple clients
- Servers can spawn clients
- Clients can join servers
- Servers can join servers
- Need **vector clocks** to ensure the **join condition**

Pro:	Better scalability; No blocking
Contra:	Complex

Contribution

- Unique use of *revision diagrams* to determine **both arbitration and visibility**.

- *Revision Diagrams* are **simple to construct** and can be **visualized easily**.
  - This eases system implementation and understanding.

Future Work and Impact

- Extend study of this programming model.
- Are there **stronger consistency guarantees** possible for subclasses of eventual consistent transactions?
- This work had an **impact** on:
  - *Cloud types for eventual consistency* [1]
    - Proposes the use of specialized cloud data types.
  - *Library abstraction for C/C++ concurrency* [2]
    - Proposes a criterion for sound library abstraction in the new C11 and C++11 memory model.

[1] Microsoft Research
[2] Mark Batty, Mike Dodds, Alexey Gotsman