ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Asynchronous Systems

Christoph Koch
School of Computer & Communication Sciences, EPFL

Synchrony

- Synchronous communication / computation: Computation proceeds in distinct aligned epochs/rounds across the distributed system.
  - *Example: Spark; BSP*
  - *Requires synchronization method to ensure an epoch is completed on all machines and machines progress to next epoch together – consensus.*
- Asynchronous computation
  - *Every node progresses individually, communication by message passing.*
  - *Example: certain peer-to-peer systems; gossiping systems (e.g. at Amazon)*
- Asynchrony is the default – it takes an effort to achieve synchronous computation.

Consistency – Distributed Key-Value Store Case

- A distributed, replicated store: All nodes contain all key-value pairs.
  - *Distribution to serve more reads.*
  - *When a value is updated, it needs to be written to all nodes.*
  - *To read a value by key, contact one node and request the value.*

Notions of consistency:

1. If I read the value for the same key repeatedly (potentially from different nodes), I never read an older version than I read before.
2. If I read the value for the same key repeatedly and I don’t modify it inbetween, I repeatedly read the same value.
3. Consistency: I receive the current/latest value, no matter which node I contact.
4. Consistency: (If there are no updates happening between two reads,) I receive the same value, no matter which node I contact.
5. …

Consistency in general

- There seem to be multiple relevant notions of consistency, and some are extremely expensive to enforce:

(If there are no updates happening between two reads,) I receive the same value, no matter which node I contact.

- Requires atomic writing; no reads while the distributed system gets updated.

- But this is only for key-value stores: how to ensure consistency in general, if there is behavior/application semantics?

A Consistency Zoo

(from Bailis et al, “Highly Available Transactions: Virtues and Limitations”. PVLDB 7(3): 181-192 (2013))

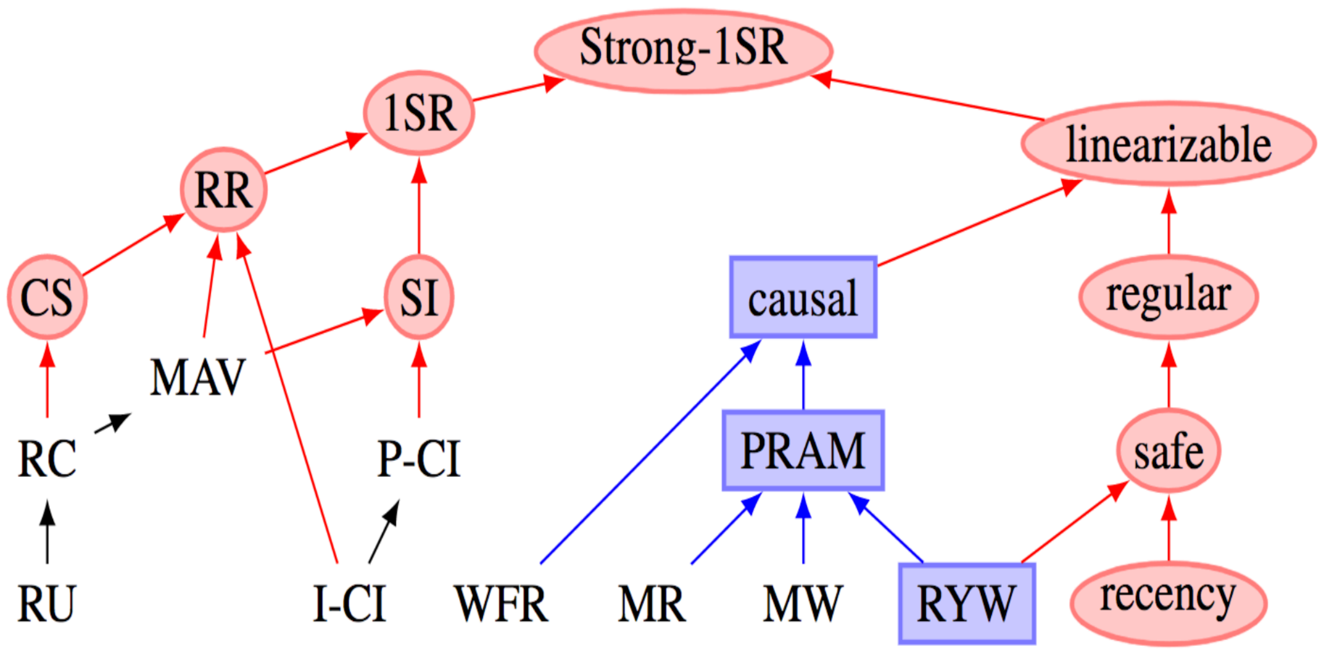


Figure 2: Partial ordering of HAT, sticky available (in boxes, blue), and unavailable models (circled, red) from Table 3. Directed edges represent ordering by model strength. Incomparable models can be simultaneously achieved, and the availability of a combination of models has the availability of the least available individual model.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Transactions: Overview

Christoph Koch
School of Computer & Communication Sciences, EPFL

Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
- Users want to access database concurrently.

Abstraction: A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.

- *A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned with what data is read/ written from/to the database.*

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself (isolation).
- Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
  - *DBMS will enforce some integrity constraints.*
  - *Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).*

Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic.* That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - *DBMS logs all actions so that it can undo the actions of aborted transactions.*

The ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

Example

- Consider two transactions (*Xacts*):

```
T1:  BEGIN   A=A+100,   B=B-100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

- Consider a possible interleaving (*schedule*):

```
T1: A=A+100,                B=B-100
T2:          A=1.06*A,               B=1.06*B
```

- This is OK. But what about:

```
T1: A=A+100,                         B=B-100
T2:          A=1.06*A, B=1.06*B
```

- The system’s view of the second schedule:

```
T1: R(A), W(A),                          R(B), W(B)
T2:             R(A), W(A), R(B), W(B)
```

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

```
T1: R(A), W(A),                       R(B), W(B), Abort
T2:                 R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1: R(A),                       R(A), W(A), C
T2:           R(A), W(A), C
```

- Overwriting Uncommitted Data (WW Conflicts):

```
T1: W(A),                       W(B), C
T2:           W(A), W(B), C
```

Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone.
- Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
  - *Or: If Ti writes an object, Tj can read this only after Ti commits.*
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded.
  - *This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.*

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Distributed commits

Christoph Koch
School of Computer & Communication Sciences, EPFL

Two-Phase Commit

Coordinator	**Subordinate**
Send prepare	
	Force-write prepare record
	Send yes or no
Wait for all responses	
Force-write commit or abort	
Send commit or abort	
	Force-write abort or commit
	Send ACK
Wait for all ACKs	
Write end record	

Comments on 2PC

- Two rounds of communication: First, voting; then, termination. Both initiated by coordinator.
- Any site can decide to abort an Xact.
- Every msg reflects a decision by the sender; to ensure that this decision survives failures, it is first recorded in the local log.
- All commit protocol log recs for an Xact contain Xactid and Coordinatorid. The coordinator’s abort/commit record also includes ids of all subordinates.

Restart After a Failure at a Site

- If we have a commit or abort log rec for Xact T, but not an end rec, must redo/undo T.
  - *If this site is the coordinator for T, keep sending **commit/abort** msgs to subs until **acks** received.*
- If we have a prepare log rec for Xact T, but not commit/abort, this site is a subordinate for T.
  - *Repeatedly contact the coordinator to find status of T, then write **commit/abort** log rec; redo/undo T; and write **end** log rec.*
- If we don’t have even a prepare log rec for T, unilaterally abort and undo T.
  - *This site may be coordinator! If so, subs may send msgs.*

Blocking

- If coordinator for Xact T fails, subordinates who have voted yes cannot decide whether to commit or abort T until coordinator recovers.
  - *T is blocked.*
  - *Even if all subordinates know each other (extra overhead in prepare msg) they are blocked unless one of them voted no.*

Link and Remote Site Failures

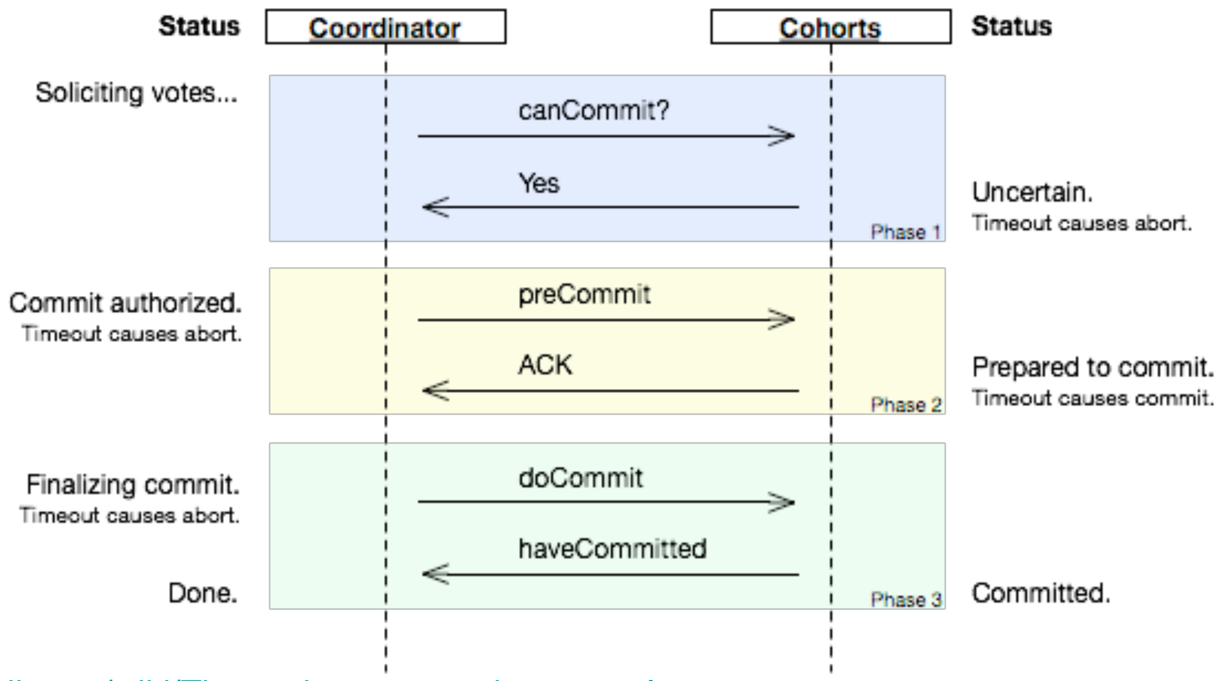
- If a remote site does not respond during the commit protocol for Xact T, either because the site failed or the link failed:
  - *If the current site is the coordinator for T, should abort T.*
  - *If the current site is a subordinate, and has not yet voted yes, it should abort T.*
  - *If the current site is a subordinate and has voted yes, it is blocked until the coordinator responds.*

Observations on 2PC

- Ack msgs used to let coordinator know when it can “forget” an Xact; until it receives all acks, it must keep T in the Xact Table.
- If coordinator fails after sending prepare msgs but before writing commit/ abort log recs, when it comes back up it aborts the Xact.
- If a subtransaction does no updates, its commit or abort status is irrelevant.

Impossibility result; 3PC

- 2PC cannot recover from the failure of both the leader and a worker.
- 3PC can (but cannot deal with network partitions):



https://en.wikipedia.org/wiki/Three-phase_commit_protocol

General consensus algorithms; impossibility

- Paxos – see https://en.wikipedia.org/wiki/Paxos_(computer_science)

- Assumptions: fully asynchronous system
  - *Processes can fail-stop (crash failures); but no Byzantine behavior.*
  - *Messages can be lost or be delayed arbitrarily.*
- No algorithm for consensus can exist!
- Proof assumes worst-case timing.
- Randomized delay messaging “solves” the problem.

Crash Failures vs. Byzantine Failures

- Failures: Simple crash failures vs. adversarial (Byzantine) failures
  - *Byzantine: a node may run a different protocol or be malicious -- explicitly try to do the worst thing possible to defeat the agreed-upon protocol.*
- Byzantine fault tolerance – consensus **impossible** unless more than a third of the involved nodes are honest.
- Commander and lieutenants problem
  - *Lieutenants will do what commander says until he's a traitor. Example: Two lieutenants; traitorous commander sends "attack" to one, "retreat" to the other.*
  - *Lieutenants exchange the command they receive, but cannot determine if the commander or the other lieutenant is the traitor.*
  - *Fix: more than two lieutenants: lieutenants can do a quorum, assuming there is at most one traitor.*
- In general, need 3f+1 nodes, f is number of Byzantine nodes. Also requires bounded delay messaging (~synchronous communication).

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Impossibility to Scale Out Transaction Processing

Christoph Koch
School of Computer & Communication Sciences, EPFL

Two-Phase commit/consensus does not scale

- No consensus without at least a communication round-trip (from a master, in 2PC) to each node.
- As #nodes increases, distance between nodes increases. Roundtrip latency increases (can’t beat the speed of light).
- Transaction throughput upper-bounded by 1/latency.
- Example: If consensus takes 1ms, can’t do more than 1000 Xacts/s.

- For current hardware: as #nodes goes into the 100s, transaction throughput decreases as you add more nodes.

Cf. Amdahl’s law

CAP Theorem (Brewer; Gilbert&Lynch)

- Consistency: Possible answers to read request: either newest value or error
- Availability: Possible answers to read request: a value, not necessarily the newest.
- Partition Tolerance: System still works if some of the nodes are unreachable.
- CAP Theorem: Can’t have Consistency, Availability, and Partition Tolerance all at once in a distributed key-value store.
  - *In other words: In a real network (which can fail -> partitions), can’t have both consistency and availability.*
- Can have any two of C,A,P, but not all three.
  - *Choose C+P: strongly consistent systems (e.g. ACID DBMS)*
  - *Choose A+P: eventually consistent systems (e.g. NoSQL systems, Dynamo)*
  - *Choose C+A: single-node system.*

Consistency vs. Latency (PACELC) “Theorem”

- Latency is similar to a partition.
- Follows from CAP Theorem if you consider every network partition temporary (a request may block if a node cannot be reached).
- Even in case of no partitions, contacting other machines for consensus takes time, like a network partition.

“PACELC Theorem”: Consistency-Latency tradeoff.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Weak Consistency

Christoph Koch

School of Computer & Communication Sciences, EPFL

Eventual Consistency

Assumptions

- Distributed key-value store
- Client-server

- There are reads and writes, but no notion of computation. No transactions.
- Atomicity is only a tool to support consistency in the presence of replicas.

Notions of weak consistency

- Causal consistency
- Read-your-writes consistency
- Session consistency
- Monotonic read consistency
- Monotonic write consistency

Eventual consistency = If there are no further chances, eventually, all reads will return the same (=last updated) value.

A system that has achieved this consistent state is said to have *converged*.

Types of “eventual” consistency (Vogels)

- **Causal consistency.** If process A has communicated to process B that it has updated a data item, a subsequent access by B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C that has no causal relationship to process A is subject to the normal eventual consistency rules.
- **Read-your-writes consistency.** Process A, after it has updated a data item, always accesses the updated value and will never see an older value.
- **Session consistency.** A process accesses the storage system in the context of a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session needs to be created and the guarantees do not overlap the sessions.
- **Monotonic read consistency.** If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.
- **Monotonic write consistency.** In this case the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

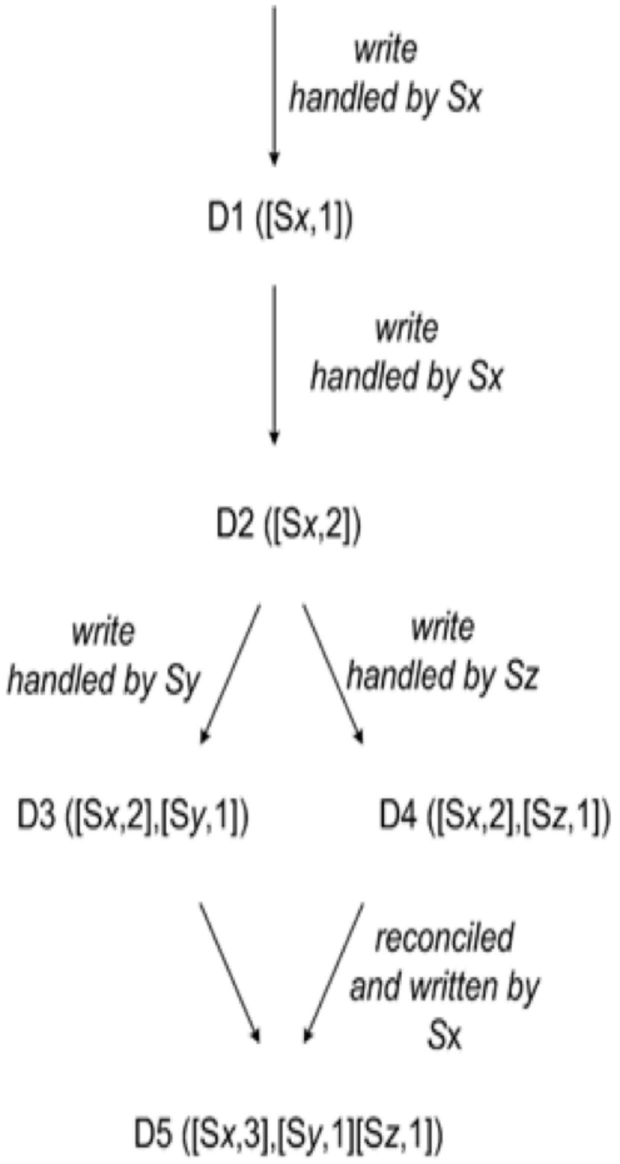
The server-side viewpoint

- N = #nodes that store replicas of the data
- W = #replicas that need to acknowledge receipt of the update before the update completes
- R = #replicas contacted when a data object is accessed through a read operation.

- W+R > N: strong consistency (quorum consensus method)
  - *N=3, R=2, W=2 – good performance/availability tradeoff*
  - *Optimize for read performance. N, W large, R=1*
- W+R <= N: weak/eventual consistency (R=1 reasonable)

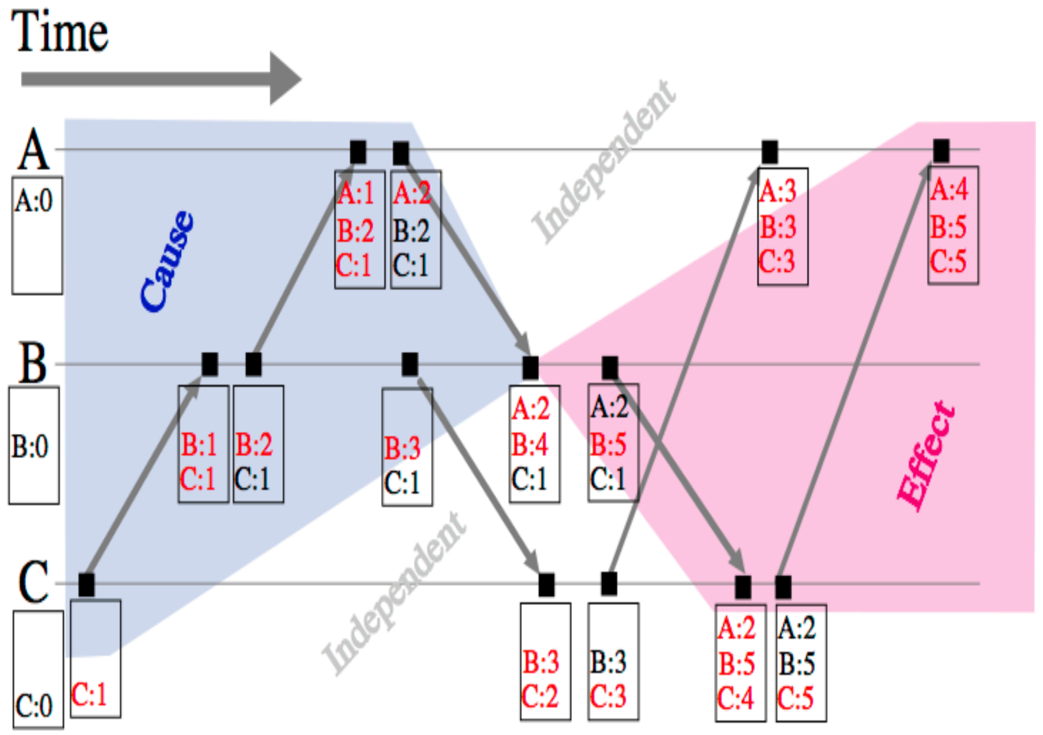
Causal consistency and vector clocks

- Causal consistency: One (relatively strong) form of eventual consistency.
- A system provides causal consistency if memory operations that potentially are causally related are seen by every node of the system in the same order.
- Enforce by vector clocks
- Application-specific conflict resolution
- Example: Amazon's Dynamo



Causal Consistency

- A system provides causal consistency if memory operations that potentially are causally related are seen by every node of the system in the same order.
- Causality by vector clocks (img source: Wikipedia):



ACID vs. weak consistency

- Cons of ACID systems: strong consistency (2PC, Paxos) does not scale.
  - *J. Gray: beyond f(hardware tech) nodes, adding more nodes makes system (transaction) throughput decrease*
- Programming weakly consistent systems is very hard and error-prone.
  - *Programming apps using ACID transactions is foolproof.*
- Few new systems use weak/eventual consistency.
- Google, initially a militant partisan to NoSQL, is now building ACID systems.
  - *Bigtable (2006, atomic tuple updates) => MegaStore (2011, entity groups) => Spanner (2012, ACID) => F1 (2012, SQL RDBMS)*