

# Distributed Systems

## 12. Distributed Transactions

Paul Krzyzanowski

[pxk@cs.rutgers.edu](mailto:pxk@cs.rutgers.edu)

# Atomic Transactions

---

- **Transaction**
  - An operation composed of a number of discrete steps.
- All the steps must be completed for the transaction to be **committed**. The results are made permanent.
- Otherwise, the transaction is **aborted** and the state of the system **reverts** to what it was before the transaction started.

# Example

---

- Buying a house:
  - Make an offer
  - Sign contract
  - Deposit money in escrow
  - Inspect the house
  - Critical problems from inspection?
  - Get a mortgage
  - Have seller make repairs
  - **Commit**: sign closing papers & transfer deed
  - **Abort**: return escrow and revert to pre-purchase stat

*All or nothing property*

# Basic Operations

---

- Transaction primitives:
  - **Begin transaction**: mark the start of a transaction
  - **End transaction**: mark the end of a transaction; try to commit
  - **Abort transaction**: kill the transaction, restore old values
  - Read/write data from files (or object stores): data will have to be restored if the transaction is aborted.

# Another Example

---

Book a flight from Newark, New Jersey to Inyokern, California. No non-stop flights are available:

Transaction begin

1. Reserve a seat for Newark to Denver (EWR→DEN)
2. Reserve a seat for Denver to Los Angeles (DEN→LAX)
3. Reserve a seat for Denver to Inyokern (LAX→IYK)

Transaction end

If there are no seats available on the LAX→IYK leg of the journey, the transaction is aborted and reservations for (1) and (2) are undone.

# Properties of transactions: ACID

---

- **Atomic**
  - The transaction happens as a single indivisible action. Everything succeeds or else the entire transaction is rolled back. Others do not see intermediate results.
- **Consistent**
  - A transaction cannot leave the database in an inconsistent state. If the system has invariants, they must hold after the transaction. E.g., total amount of money in all accounts must be the same before and after a “transfer funds” transaction.
- **Isolated (Serializable)**
  - Transactions cannot interfere with each other  
If transactions run at the same time, the final result must be the same as if they executed in some serial order.
- **Durable**
  - Once a transaction commits, the results are made permanent. No failures after a commit will cause the results to revert.

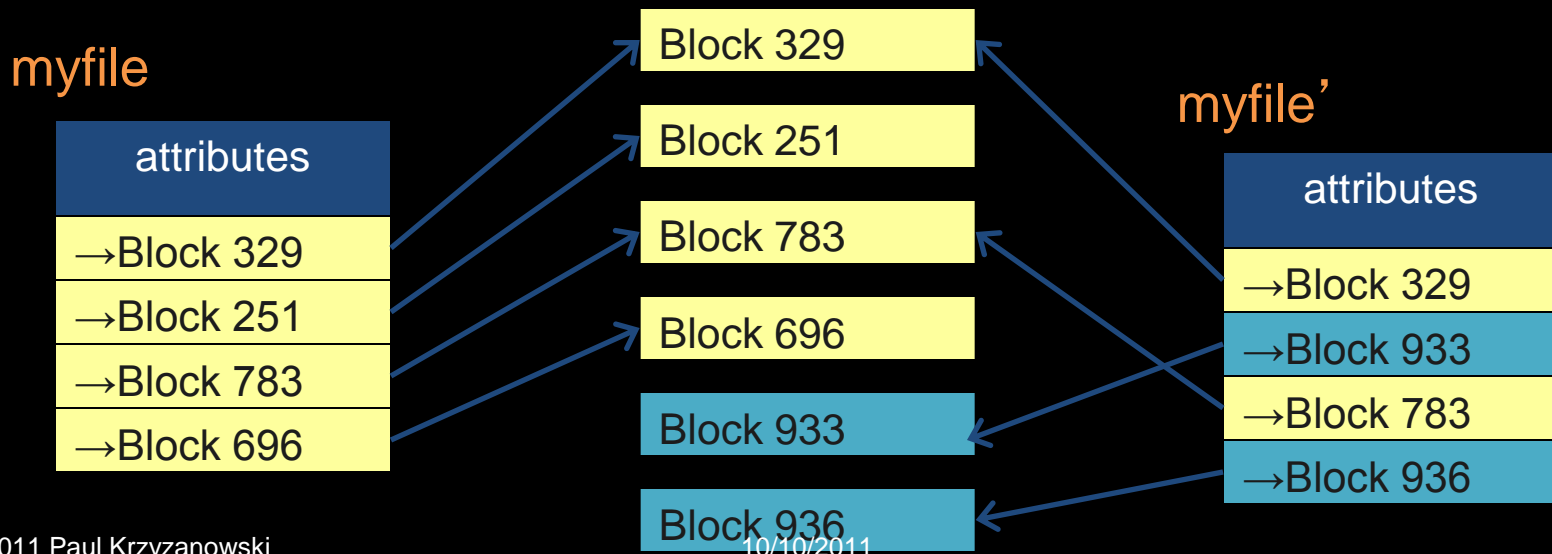
# Nested Transactions

---

- A top-level transaction may create subtransactions
- Problem:
  - subtransactions may commit (results are durable) but the parent transaction may abort.
- One solution: **private workspace**
  - Each subtransaction is given a private copy of every object it manipulates. On commit, the private copy displaces the parent's copy (which may also be a private copy of the parent's parent)

# Implementing a private workspace [PLAN A]

- Consider a Unix-like file system:
  - duplicate the file's index (i-node)
  - Create new block when it gets modified by the transaction: **shadow block**
  - Index copy points to the shadow block
- On abort: remove shadow blocks and private index
- On commit: update parent's index with private index



# Implementing a private workspace [PLAN B]

---

- Use a **writeahead log** (intentions list)
  - Keep a log in **stable storage**
  - Before making any changes to the object, write a record to the log identifying
    - $\{ \textit{transaction ID}, \textit{object ID}, \textit{old value}, \textit{new value} \}$
  - If transaction commits:
    - Write a **commit** record onto the log
  - If transaction aborts
    - Use log to back up to the original state: **rollback**
- **Stable storage**: data persists even if the system or application crashes.

# Distributed Transactions

---

- Transaction that updates data on two or more systems
- Challenge: handle machine, software, & network failures while preserving transaction integrity.

# Distributed Transactions

---

- Each computer runs a **transaction manager**
  - Responsible for subtransactions on that system
  - Transaction managers communicate with other transaction managers
  - Performs *prepare*, *commit*, and *abort* calls for subtransactions
- Each subtransaction must agree to commit changes before the transaction can complete

# Two-phase commit protocol

---

## Goal:

Reliably agree to commit or abort a collection of subtransactions

- All processes in the transaction will agree to commit or abort.
- One transaction manager is elected as a coordinator – the rest are cohorts
- Assume:
  - write-ahead log in stable storage
  - No system dies forever
  - Systems can always communicate with each other

# Two-Phase Commit Protocol: Phase 1

---

## Voting Phase

### Coordinator

- Write *prepare to commit* to log
- Send *prepare to commit* message

- Wait for reply from each cohort

### Cohort

- Work on transaction
- Wait for message from coordinator

- Receive *prepare* message.
- When ready, write *agree to commit* or *abort* to the log
- Send *agree* or *abort* reply

Get distributed agreement: the coordinator asked each cohort if it will commit or abort and received replies from each coordinator.

# Two-Phase Commit Protocol: Phase 2

## Commit Phase

### Coordinator

- Write *commit* to log
- Send *commit* or *abort*
- Wait for all cohorts to respond
- Clean up all state. Done!

### Cohort

- Wait for *commit/abort* message
- Receive *commit* or *abort*
- If a *commit* was received, write “*commit*” to the log, release all locks, update databases.
- If an *abort* was received, undo all changes
- Send *done* message

Either ask *all cohorts to commit* or *abort*  
Get everyone's response that they're done.

# What's wrong with the 2PC protocol?

---

- Biggest problem: it's a blocking protocol
- Modify 2PC to place an upper limit on the time before a transaction commits or aborts

## Three-Phase Commit Protocol

# Three-Phase Commit Protocol

---

- Same setup as the two-phase commit protocol:
  - Coordinator & Cohorts
- Add timeouts to each phase that result in an abort
- **Phase 1: *voting phase***
  - Coordinator sends *canCommit?* queries to cohorts. Get responses.
  - If *failure, timeout, or any NO replies*, then send abort to all cohorts.
  - Timeout at the cohort causes an *abort*.
  - Else continue to phase 2.
- **Phase 2: *prepare to commit phase***
  - Send a *preCommit* message to all cohorts. Get *ack* messages from all cohorts.
  - Timeout at the cohort causes the cohort to commit.
- **Phase 3: *finalize***
  - Send *doCommit* messages to cohorts. Get responses from all.

# Scaling Transactions

---

- Transactions require locking part of the database so that everyone sees consistent data at all times
  - Good on a small scale.
    - Low transaction volumes: getting multiple databases consistent is easy
  - Difficult to do efficiently on a huge scale
- Instead: *Use cached data*
- Problems / side-effects:
  - Users run the risk of seeing stale data
  - The “I” of ACID may be violated
    - E.g., two users might try to buy the last book on Amazon

# Scaling also affects availability

---

- One database with 99.9% availability
  - 8 hours, 45 minutes, 35 seconds downtime per year
- If a transaction uses 2PC protocol and requires two databases, each with a 99.9% availability:
  - Total availability =  $(0.999 \times 0.999) = 99.8\%$
  - 17 hours, 31 minutes, 12 seconds downtime per year
- If a transaction requires 5 databases:
  - Total availability = 99.5%
  - 1 day, 19 hours, 48 minutes, 0 seconds downtime per year

# Delays hurt

---

- The delays to achieve consistency can hurt business
- Amazon:
  - 0.1 second increase in response time will cost them 1% of sales
- Google:
  - 0.5 second increase in latency causes traffic to drop by 20%
- Latency is due to lots of factors
  - OS & software architecture, computing hardware, tight vs loose coupling, network links, geographic distribution, ...
  - We're only looking at the problems caused by the tight software coupling due to achieving the ACID model

<http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it>

<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>

# Brewer's CAP Theorem

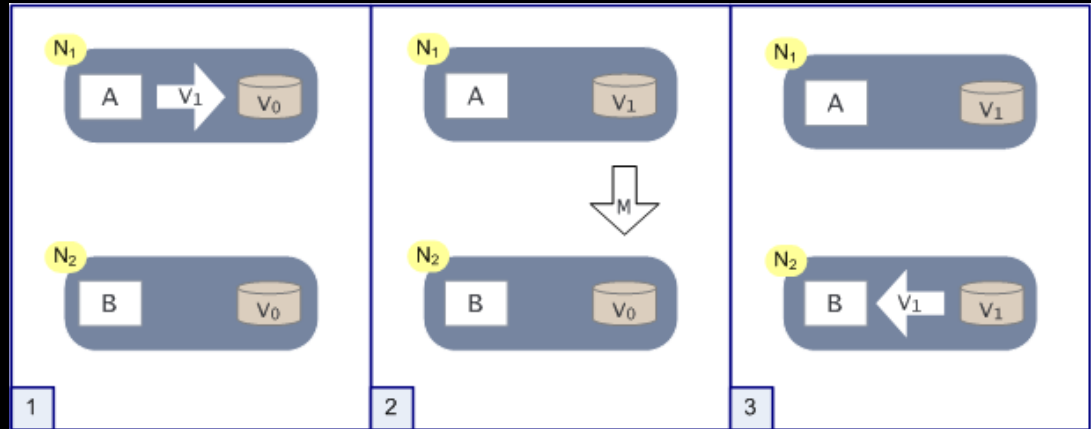
---

- Three core requirements in a shared data system:
  - Consistency
  - Availability
  - Partition Tolerance: tolerance to network partitioning
    - No set of failures less than total failure is allowed to cause the system to respond incorrectly\*
- CAP Theorem: you can only have at most two of these!

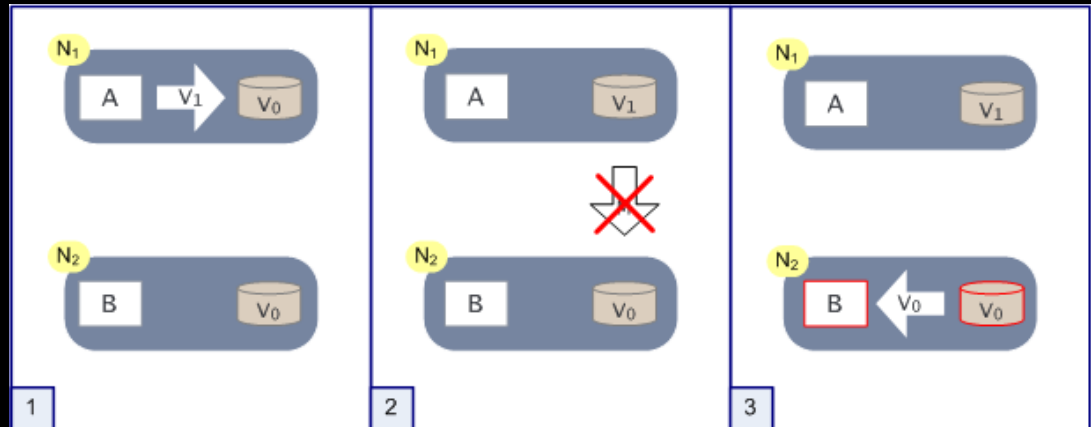
\*<http://tinyurl.com/42dt322>

# Example

- Life is good



- Network partition



From: <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>

# Giving up one of {C, A, P}

---

- Give up partitions
  - Put everything on one machine or a cluster in one rack
  - Use two-phase commit
  - Scaling suffers
- Give up availability
  - Lock data; have services wait until data is consistent
  - Classic ACID distributed databases (also 2PC)
  - Response time suffers
- Give up consistency
  - *Eventually consistent* data
  - Often use expirations/leases, queued messages for updates
  - Examples: DNS, web caching
  - Often not as bad as it looks!

# BASE: an alternative to ACID

---

- Traditional database systems want ACID
  - But scalability is a problem (lots of transactions in a distributed environment)
- Give up *Consistent* and *Isolated* in exchange for *high availability* and *high performance*
- **BASE:**
  - Basically **A**vailable
  - **S**oft-state
  - **E**ventual **C**onsistency

# ACID vs. BASE

---

## ACID

- Strong consistency
- Isolation
- Focus on *commit*
- Nested transactions
- Availability problematic
- Pessimistic access to data (locking)

## BASE

- Weak consistency (stale data at times)
- High availability
- Best effort approach
- Optimistic access to data
- Simpler model (but harder for app developer)
- Faster

From Eric Brewer's PODC Keynote, July 2000  
<http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>

# A place for BASE

---

- ACID is neither dead nor useless
  - Many environments require it
- BASE has become common for large-scale web apps
  - eBay, Twitter, Amazon
  - Eventually consistent model not always surprising to users
    - Cellphone usage data
    - Banking transactions (e.g., transferring funds or ATM withdrawal showing up on statement)
    - Posting of frequent flyer miles

The End