

Distributed Systems

11. Distributed Commit Protocols

Paul Krzyzanowski
Rutgers University
Fall 2017

October 23, 2017

© 2014-2017 Paul Krzyzanowski

1

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.
 - **rollback** = reverting to a previous state

October 23, 2017

© 2014-2017 Paul Krzyzanowski

2

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

October 23, 2017

© 2014-2017 Paul Krzyzanowski

3

Basic Operations

Transaction primitives:

- *Begin transaction*: mark the start of a transaction
- *End transaction*: mark the end of a transaction – no more tasks
- *Commit transaction*: make the results permanent
- *Abort transaction*: kill the transaction, restore old values
- *Read/write/compute data* (modify files or objects)
 - But data will have to be restored if the transaction is aborted.

October 23, 2017

© 2014-2017 Paul Krzyzanowski

4

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.

October 23, 2017

© 2014-2017 Paul Krzyzanowski

5

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. 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 or see intermediate results 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.

October 23, 2017

© 2014-2017 Paul Krzyzanowski

6

Distributed Transactions

Transaction that updates data on two or more systems

Challenge
Handle machine, software, & network failures while preserving transaction integrity

October 23, 2017 © 2014-2017 Paul Krzyzanowski 10

Distributed Transactions

- Each computer runs a **transaction manager**
 - Responsible for sub-transactions on that system
 - Performs *prepare*, *commit*, and *abort* calls for subtransactions
- Every sub-transaction must agree to commit changes before the overall transaction can complete

October 23, 2017 © 2014-2017 Paul Krzyzanowski 11

Commits Among Subtransactions = Consensus

Remember consensus?
– Agree on a value proposed by at least one process

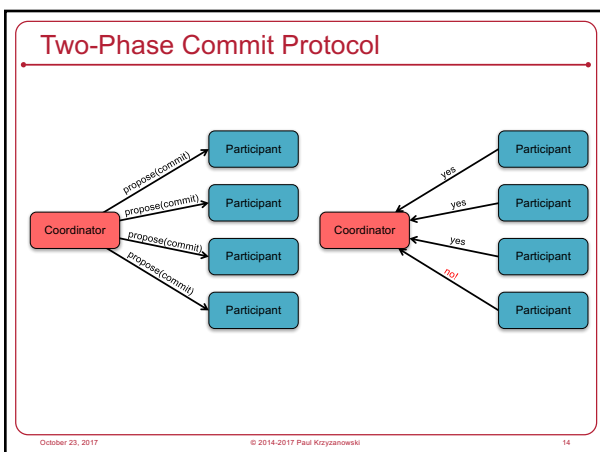
BUT – here we need unanimous agreement to commit

The coordinator proposes to commit a transaction
– **All** participants agree ⇒ all participants then commit
– **Not all** participants agree ⇒ all participants then abort

October 23, 2017 © 2014-2017 Paul Krzyzanowski 12

Two-Phase Commit Protocol

October 23, 2017 © 2014-2017 Paul Krzyzanowski 13

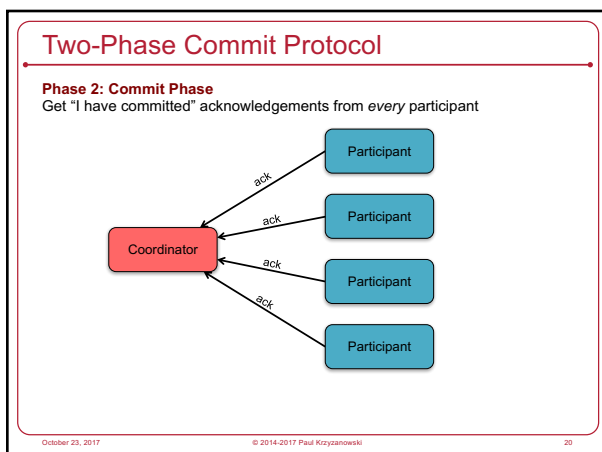
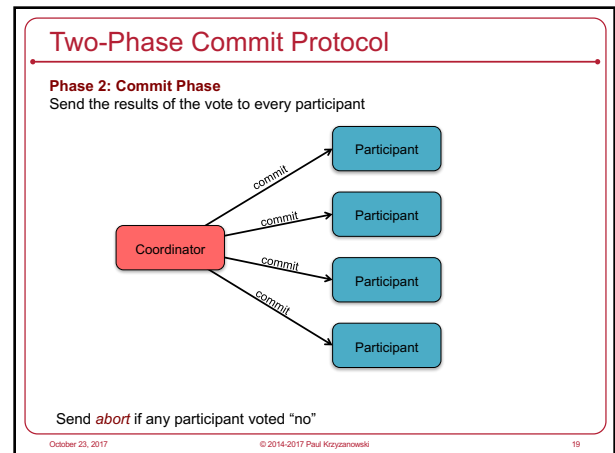
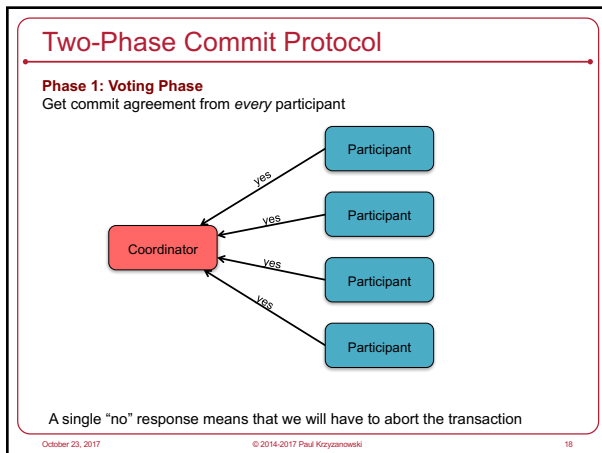
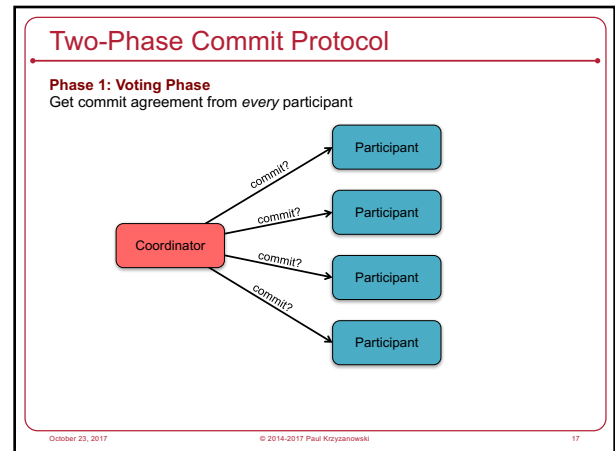
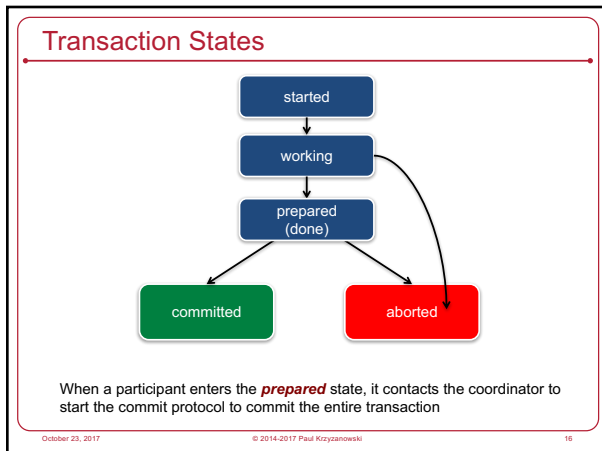


Two-phase commit protocol

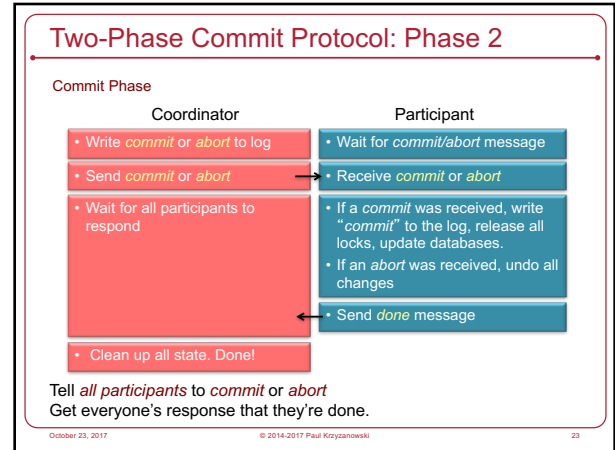
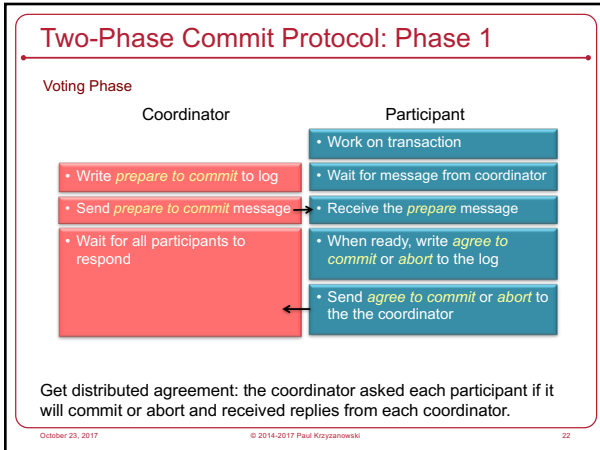
Goal:
*Reliably agree to **commit** or **abort** a collection of sub-transactions*

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

October 23, 2017 © 2014-2017 Paul Krzyzanowski 15



- ### Dealing with failure
- 2PC assumes a **fail-recover model**
 - Any failed system will eventually recover
 - A recovered system cannot change its mind
 - If a node agreed to commit and then crashed, it must be willing and able to commit upon recovery
 - Each system will use a **write-ahead (transaction) log**
 - Keep track of where it is in the protocol (and what it agreed to)
 - As well as values to enable commit or abort (rollback)
 - This enables fail-recover
- October 23, 2017 © 2014-2017 Paul Krzyzanowski 21



Dealing with failure

Failure during Phase 1 (voting)

Coordinator dies
Some participants may have responded; others have no clue
⇒ Coordinator **restarts voting**: checks log; sees that voting was in progress

Participant dies
The participant may have died before or after sending its vote to the coordinator
⇒ If the coordinator received the vote, it waits for other votes and **go to phase 2**
⇒ Otherwise: **wait for the participant** to recover and respond (keep querying it)

October 23, 2017 © 2014-2017 Paul Krzyzanowski 24

Dealing with failure

Failure during Phase 2 (commit/abort)

Coordinator dies
Some participants may have been given commit/abort instructions
⇒ **Coordinator restarts**; checks log; **informs all participants** of chosen action

Participant dies
The participant may have died before or after getting the commit/abort request
⇒ Coordinator **keeps trying to contact the participant** with the request
⇒ **Participant recovers**; checks log; gets request from coordinator
– If it committed/aborted, acknowledge the request
– Otherwise, process the commit/abort request and send back the acknowledgement

October 23, 2017 © 2014-2017 Paul Krzyzanowski 25

Adding a recovery coordinator

- Another system can take over for the coordinator
 - Could be a participant that detected a timeout to the coordinator
- Recovery node needs to find the state of the protocol
 - Contact **ALL** participants to see how they voted
 - If we get **voting results from all participants**
 - We know that Phase 1 has completed
 - If all participants voted to commit ⇒ send **commit** request
 - Otherwise send **abort** request
 - If **ANY participant states that it has not voted**
 - We know that Phase 1 has **not** completed
 - ⇒ Restart the protocol
- But ... if any participant node also crashes, we're stuck!
 - Have to wait for recovery

October 23, 2017 © 2014-2017 Paul Krzyzanowski 26

What's wrong with the 2PC protocol?

Biggest problem: it's a **blocking protocol** with failure modes that require all systems to recover eventually

- If the coordinator crashes, participants have no idea whether to commit or abort
 - A recovery coordinator helps
- If a coordinator AND a participant crashes
 - The system has no way of knowing the result of the transaction
 - It might have committed for the crashed participant – hence all others must block

When a participant gets a commit/abort message, it does not know if every other participant was informed of the result

October 23, 2017 © 2014-2017 Paul Krzyzanowski 27

Three-Phase Commit Protocol

October 23, 2017

© 2014-2017 Paul Krzyzanowski

28

Three-Phase Commit Protocol

- Same setup as the two-phase commit protocol:
 - Coordinator & Participants
- Add **timeouts** to each phase that result in an abort
- **Propagate the result of the commit/abort vote** to each participant *before* telling them to act on it
 - This will allow us to recover the state if any participant dies

October 23, 2017

© 2014-2017 Paul Krzyzanowski

29

Three-Phase Commit Protocol

Split the second phase of 2PC into two parts:

2a. "Precommit" (prepare to commit) phase

- Send *Prepare* message to all participants when it received a **yes** from all participants in phase 1
- Participants can prepare to commit but cannot do anything that cannot be undone
- Participants reply with an acknowledgement
- **Purpose:** *let every participant know the state of the result of the vote so that state can be recovered if anyone dies*

2b. "Commit" phase (same as in 2PC)

- If coordinator gets ACKs for all *prepare* messages
 - It will send a *commit* message to all participants
- Else it will abort – send an *abort* message to all participants

October 23, 2017

© 2014-2017 Paul Krzyzanowski

30

Three-Phase Commit Protocol: Phase 1

Phase 1: Voting phase

- Coordinator sends *canCommit?* queries to participants & gets responses
 - Purpose: Find out if everyone agrees to commit
- [!] If the coordinator gets a **timeout** from any participant, or **any NO replies** are received
 - Send an **abort** to all participants
- [!] If a participant times out waiting for a request from the coordinator
 - It **aborts** itself (assume coordinator crashed)
- Else continue to phase 2

We can abort if the participant and/or coordinator dies

October 23, 2017

© 2014-2017 Paul Krzyzanowski

31

Three-Phase Commit Protocol

Phase 2: Prepare to commit phase

- Send a *prepare* message to all participants.
- Get **OK** messages from all participants
- Purpose: let all participants know the decision to commit
- [!] If a participant times out: assume it crashed; send **abort** to all participants

Phase 3: Finalize phase

- Send **commit** messages to participants and get responses from all
- [!] If participant times out: **contact any other participant** and move to that state (**commit** or **abort**)
- [!] If coordinator times out: that's ok – we know what to do

October 23, 2017

© 2014-2017 Paul Krzyzanowski

32

3PC Recovery

If the coordinator crashes

A recovery node can query the state from any available participant

Possible states that the participant may report:

Already committed

- That means that every other participant has received a *Prepare to Commit*
- Some participants may have committed
 - ⇒ Send **Commit** message to all participants (just in case they didn't get it)

Not committed but received a Prepare message

- That means that all participants agreed to commit; some may have committed
- Send **Prepare to Commit** message to all participants (just in case they didn't get it)
- Wait for everyone to acknowledge; then **commit**

Not yet received a Prepare message

- This means no participant has committed; some may have agreed
- Transaction can be **aborted** or the commit protocol can be **restarted**

October 23, 2017

© 2014-2017 Paul Krzyzanowski

33

3PC Weaknesses

Main weakness of 3PC

- May have problems when the **network gets partitioned**
 - Partition A: nodes that received *Prepare* message
 - Recovery coordinator for A: allows commit
 - Partition B: nodes that did not receive *Prepare* message
 - Recovery coordinator for B: aborts
 - Either of these actions are legitimate as a whole
 - But when the network merges back, the system is inconsistent
- Not good when a crashed coordinator recovers
 - It needs to find out that someone took over and stay quiet
 - Otherwise it will mess up the protocol, leading to an inconsistent state

October 23, 2017 © 2014-2017 Paul Krzyzanowski 34

3PC coordinator recovery problem

- Suppose
 - a coordinator sent a *Prepare* message to all participants
 - all participants acknowledged the message
 - BUT the coordinator died before it got all acknowledgements
- A recovery coordinator queries a participant
 - Continues with the commit: Sends *Prepare*, gets *ACKs*, sends *Commit*
- Around the same time...**the original coordinator recovers**
 - Realizes it is still missing some replies from the *Prepare*
 - Gets timeouts from some and decides to send an *Abort* to all participants
- Some processes may commit while others abort!

• 3PC works well when servers crash (fail-stop model)
 • 3PC is not resilient against fail-recover environments

October 23, 2017 © 2014-2017 Paul Krzyzanowski 35

Paxos Commit

October 23, 2017 © 2014-2017 Paul Krzyzanowski 36

What about Paxos?

- Interface to Paxos
 - Client proposes a value and sends it to the Paxos leader
 - Paxos acceptors will send out the totally-ordered value
- What does Paxos consensus offer?
 - Total ordering of proposals
 - **Fault tolerance**: proposal is accepted if a **majority** of acceptors accept it
 - There is always enough data available to recover the state of proposals
 - Is provably resilient in asynchronous networks
- Paxos-based commit is a generalization of 2PC
 - Use multiple coordinators to avoid blocking if the coordinator fails
 - Run a consensus algorithm on the commit/abort decision of EACH participant

October 23, 2017 © 2014-2017 Paul Krzyzanowski 37

Paxos for Commit Fault-tolerant Coordinator

The cast:

- One instance of Paxos per participant (N participants)
- Set of $2F+1$ acceptors and a leader play the role of the coordinator
 - We can withstand the failure of F acceptors
 - Leader = proposer elected to be in charge

```

Ready to start Participant → begin commit → Leader
Tell everyone Leader → prepare → { Participanti=1..N }
Each instance of Paxos proposes to commit or abort Participanti=1..N → value = (prepared | aborted) → { Acceptors }
Each instance of Paxos tells the result to the leader { Acceptors } → Leader
    
```

A leader will get at least $F+1$ messages for each instance
 Commit iff every participant's instance of Paxos chooses *Prepared*
 Paxos commit = 2PC if one acceptor

October 23, 2017 © 2014-2017 Paul Krzyzanowski 38

Virtual Synchrony vs. Transactions vs. Paxos

- Virtual Synchrony
 - Fastest & most scalable
 - Focuses on group membership management & reliable multicasts
- Two-Phase & Three-Phase Commit
 - Most expensive – requires extensive use of stable storage
 - Designed for transactional activities
 - Three-phase commit handles coordinator failure
 - Not suitable for high speed messaging
- Paxos
 - Performance limited by its two-phase protocol
 - Great for consensus & fault tolerance
 - Adds ordering of proposals over 2PC

October 23, 2017 © 2014-2017 Paul Krzyzanowski 39

Scaling & Consistency

October 23, 2017 © 2014-2017 Paul Krzyzanowski 40

Reliance on multiple systems 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

October 23, 2017 © 2014-2017 Paul Krzyzanowski 41

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
- Add replication – processes can read any replica
 - **But all replicas must be locked during updates to ensure consistency**
- Risks of not locking:
 - 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

October 23, 2017 © 2014-2017 Paul Krzyzanowski 42

Delays hurt

- The delays to achieve consistency can hurt business
- Amazon:
 - 0.1 second increase in response time costs 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-crash-it>
<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>

October 23, 2017 © 2014-2017 Paul Krzyzanowski 43

Eric Brewer's CAP Theorem

Three core requirements in a shared data system:

1. **Atomic, Isolated Consistency**
 - Operations must appear totally ordered and each is isolated
2. **Availability**
 - Every request received by a non-failed node must result in a response
3. **Partition Tolerance**: tolerance to network partitioning
 - Messages between nodes may be lost

No set of failures less than total failure is allowed to cause the system to respond incorrectly*

CAP Theorem: when there is a network partition, you cannot guarantee both availability & consistency

Commonly stated as *you can have at most two of the three: C, A, or P*

*goo.gl/7nsj1R

October 23, 2017 © 2014-2017 Paul Krzyzanowski 44

Example: Partition

Life is good

A writes v_1 on N_1 .
 v_1 propagates to N_2 .
 B reads v_1 on N_2 .

Network partition occurs

A writes v_1 on N_1 .
 v_1 cannot propagate to N_2 .
 B reads v_0 on N_2 .

Read old value or wait?

Do we want to give up consistency or availability?

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

October 23, 2017 © 2014-2017 Paul Krzyzanowski 45

Giving up one of {C, A, P}

- **Ensure partitions never occur**
 - Put everything on one machine or a cluster in one rack: high availability clustering
 - Use two-phase commit or three phase commit
 - **Scaling suffers**
- **Give up availability** [system is consistent & can handle partitioning]
 - Lock data: have services wait until data is consistent
 - Classic ACID distributed databases (also 2PC)
 - **Response time suffers**
- **Give up consistency** [system is available & can handle partitioning]
 - **Eventually consistent** data
 - Use expirations/leases, queued messages for updates
 - Examples: DNS, web caching, Amazon Dynamo, Cassandra, CouchDB
 - Often not as bad as it sounds!

We really want partition tolerance & high availability for a distributed system!

October 23, 2017 © 2014-2017 Paul Krzyzanowski 46

Partitions will happen

- With distributed systems, **we expect partitions to occur**
 - Maybe not full failure but high latency can act like a failure
 - The CAP theorem says we have a tradeoff between availability & consistency
- We'd like **availability and consistency**
 - We get availability via **replication**
 - We get consistency with **atomic updates**
 1. Lock all copies before an update
 2. Propagate updates
 3. Unlock
- We can choose **high availability**
 - Allow reads before all nodes are updated (avoid locking)
- or choose **consistency**
 - Enforce proper locking of nodes for updates

October 23, 2017 © 2014-2017 Paul Krzyzanowski 47

Eventual Consistency

- 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**
 - Get rid of locking in exchange for multiple versions
 - Incremental replication
- **BASE:**
 - Basically Available
 - Soft-state
 - Eventual Consistency
- **Consistency model**
 - If no updates are made to a data item, *eventually* all accesses to that item will return the last updated value

October 23, 2017 © 2014-2017 Paul Krzyzanowski 48

ACID vs. BASE

ACID	BASE
• Strong consistency	• Weak (eventual) consistency: stale data at times
• Isolation	• High availability
• Focus on <i>commit</i>	• Best effort approach
• Nested transactions	• Optimistic access to data
• Availability can suffer	• Simpler model (but harder for app developer)
• Pessimistic access to data (locking)	• Faster

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

October 23, 2017 © 2014-2017 Paul Krzyzanowski 49

A place for BASE

- ACID is neither dead nor useless
 - Many environments require it
- BASE has become common for large-scale web apps where replication & fault tolerance is crucial
 - eBay, Twitter, Amazon
 - Eventually consistent model not always surprising to users
 - Cellphone usage data
 - Banking transactions (e.g., fund transfer activity showing up on statement)
 - Posting of frequent flyer miles

But ... the app developer has to worry about update conflicts and reading stale data

October 23, 2017 © 2014-2017 Paul Krzyzanowski 50

The end

October 23, 2017 © 2014-2017 Paul Krzyzanowski 51