

## Distributed Systems

### 11. Distributed commit protocols

Paul Krzyzanowski  
Rutgers University  
Fall 2016

November 4, 2016

© 2014-2016 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

November 4, 2016

© 2014-2016 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 state

*All or nothing property*

November 4, 2016

© 2014-2016 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.

November 4, 2016

© 2014-2016 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.

November 4, 2016

© 2014-2016 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. Failures after a commit will cause the results to revert.

November 4, 2016

© 2014-2016 Paul Krzyzanowski

6

### Distributed Transactions

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

November 4, 2016 © 2014-2016 Paul Krzyzanowski 10

### 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
- Every subtransaction must agree to commit changes before the overall transaction can complete

November 4, 2016 © 2014-2016 Paul Krzyzanowski 11

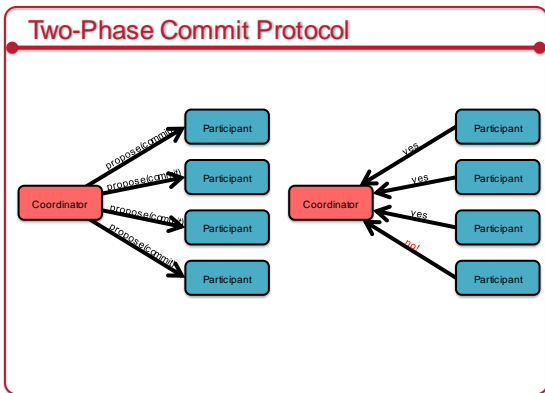
### Commits Among Subtransactions = Consensus

- Remember consensus?
  - Agree on a value proposed by at least one process
- The coordinator proposes to commit a transaction
  - **All** participants agree ⇒ all participants then commit
  - **Not all** participants agree ⇒ all participants then abort
- Here, we need **unanimous** agreement to commit

November 4, 2016 © 2014-2016 Paul Krzyzanowski 12

### Two-Phase Commit Protocol

November 4, 2016 © 2014-2016 Paul Krzyzanowski 13



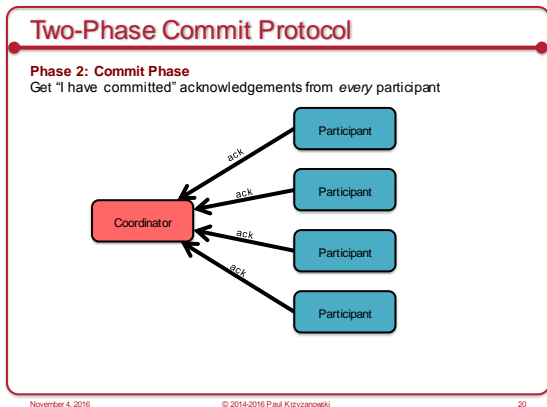
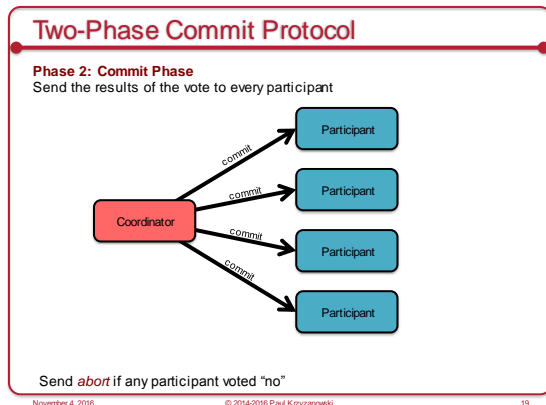
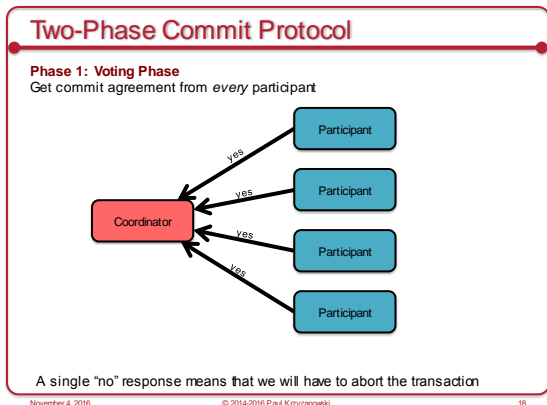
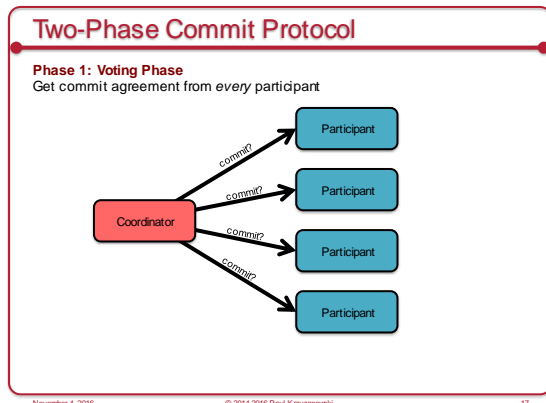
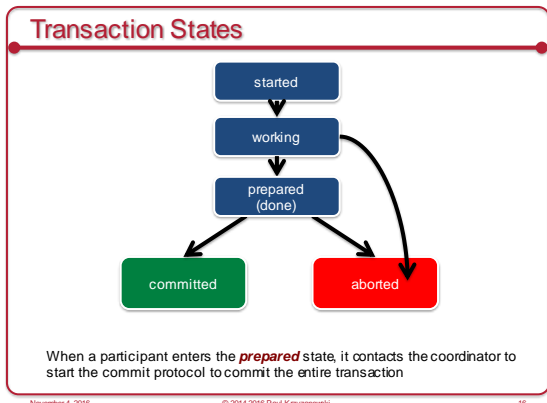
November 4, 2016 © 2014-2016 Paul Krzyzanowski 14

### Two-phase commit protocol

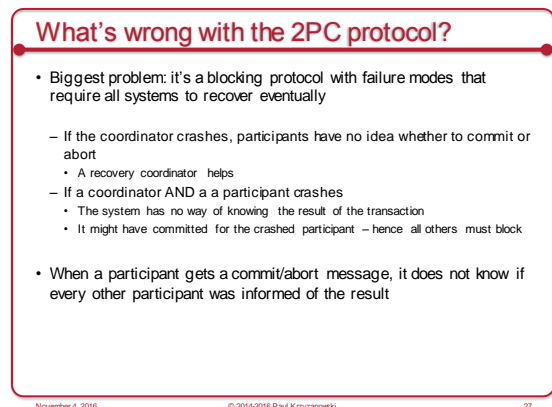
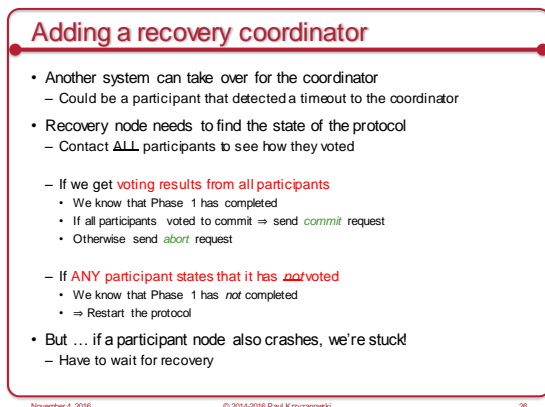
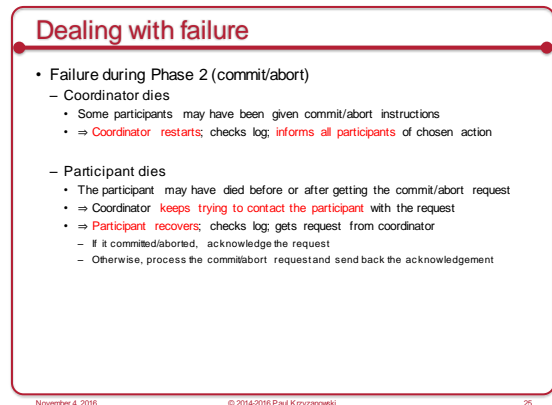
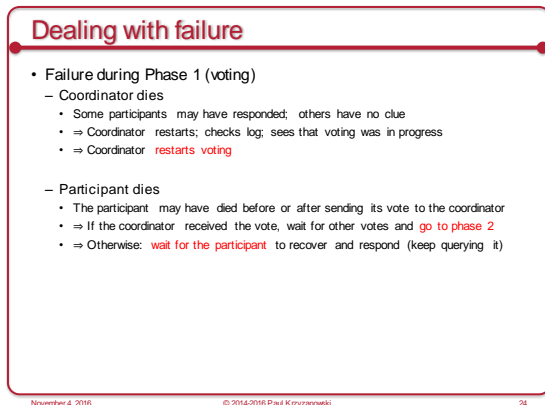
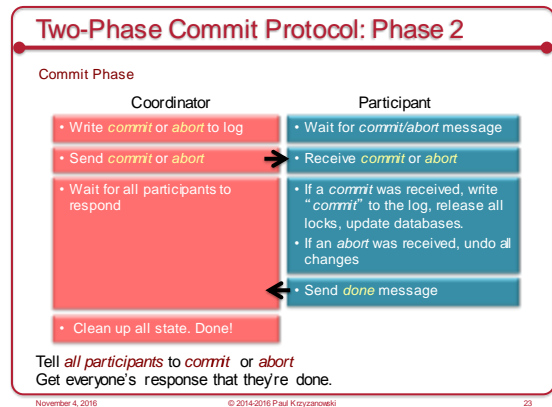
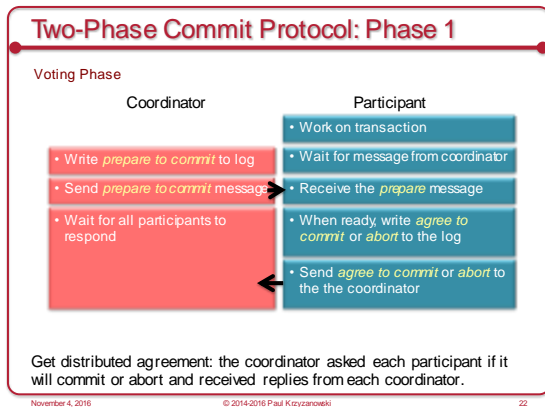
Goal:  
*Reliably agree to **commit** or **abort** a collection of sub-transactions*

- **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

November 4, 2016 © 2014-2016 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
- November 4, 2016 © 2014-2016 Paul Krzyzanowski 21



## Three-Phase Commit Protocol

November 4, 2016

© 2014-2016 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

November 4, 2016

© 2014-2016 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

November 4, 2016

© 2014-2016 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

November 4, 2016

© 2014-2016 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

November 4, 2016

© 2014-2016 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**

November 4, 2016

© 2014-2016 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

### 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**

### Paxos Commit

### 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
  - Run a consensus algorithm on the commit/abort decision of EACH participant

### 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  $F+1$  acceptors
  - Leader = proposer elected to be in charge

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

### 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
  - 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

# Scaling & Consistency

November 4, 2016 © 2014-2016 Paul Krzyzanowski 40

# Scaling Transactions

- Transactions require locking part of the database so that every one 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
  - 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

November 4, 2016 © 2014-2016 Paul Krzyzanowski 41

# 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 =  $(0.999 * 0.999) = 99.8\%$
  - 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

November 4, 2016 © 2014-2016 Paul Krzyzanowski 42

# 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, tightvs 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/browsers-cap-theorem>

November 4, 2016 © 2014-2016 Paul Krzyzanowski 43

# Eric Brewer's CAP Theorem

- Three core requirements in a shared data system:
1. Consistency
  2. Availability
  3. 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>

November 4, 2016 © 2014-2016 Paul Krzyzanowski 44

# Example

Life is good

Writes  $v_1$  on  $N_1$ .  
 $v_1$  propagates to  $N_2$ .  
 Reads  $v_1$  on  $N_3$ .

Network partition occurs

Writes  $v_1$  on  $N_1$ .  
 $v_1$  cannot propagate to  $N_3$ .  
 Reads  $v_1$  on  $N_2$ .

Do we want to give up consistency or availability?

Read old value or wait?

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

November 4, 2016 © 2014-2016 Paul Krzyzanowski 45

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

- Give up partitions [partitions cannot 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 may not be consistent]
  - Eventually consistent data
  - Use expirations/leases, queued messages for updates
  - Examples: DNS, web caching, Amazon Dynamo, Hadoop HBase, CouchDB
  - Often not as bad as it sounds!

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

November 4, 2016 © 2014-2016 Paul Krzyzanowski 46

### 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*
  - Get rid of locking in exchange for multiple versions
  - Incremental replication
- BASE:
  - Basically Available
  - Soft-state
  - Eventual Consistency

November 4, 2016 © 2014-2016 Paul Krzyzanowski 47

### ACID vs. BASE

ACID	BASE
<ul style="list-style-type: none"> <li>• Strong consistency</li> <li>• Isolation</li> <li>• Focus on <i>commit</i></li> <li>• Nested transactions</li> <li>• Availability is problematic</li> <li>• Pessimistic access to data (locking)</li> </ul>	<ul style="list-style-type: none"> <li>• Weak (eventual) consistency: stale data at times</li> <li>• High availability</li> <li>• Best effort approach</li> <li>• Optimistic access to data</li> <li>• Simpler model (but harder for app developer)</li> <li>• Faster</li> </ul>

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

November 4, 2016 © 2014-2016 Paul Krzyzanowski 48

### 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

November 4, 2016 © 2014-2016 Paul Krzyzanowski 49

# The End

November 4, 2016 © 2014-2016 Paul Krzyzanowski 50