MDCC : MULTI-DATA CENTER CONSISTENCY

Authors: Tim Kraska, Gene Pang, Michael Franklin, Samuel Madden, Alan Fekete

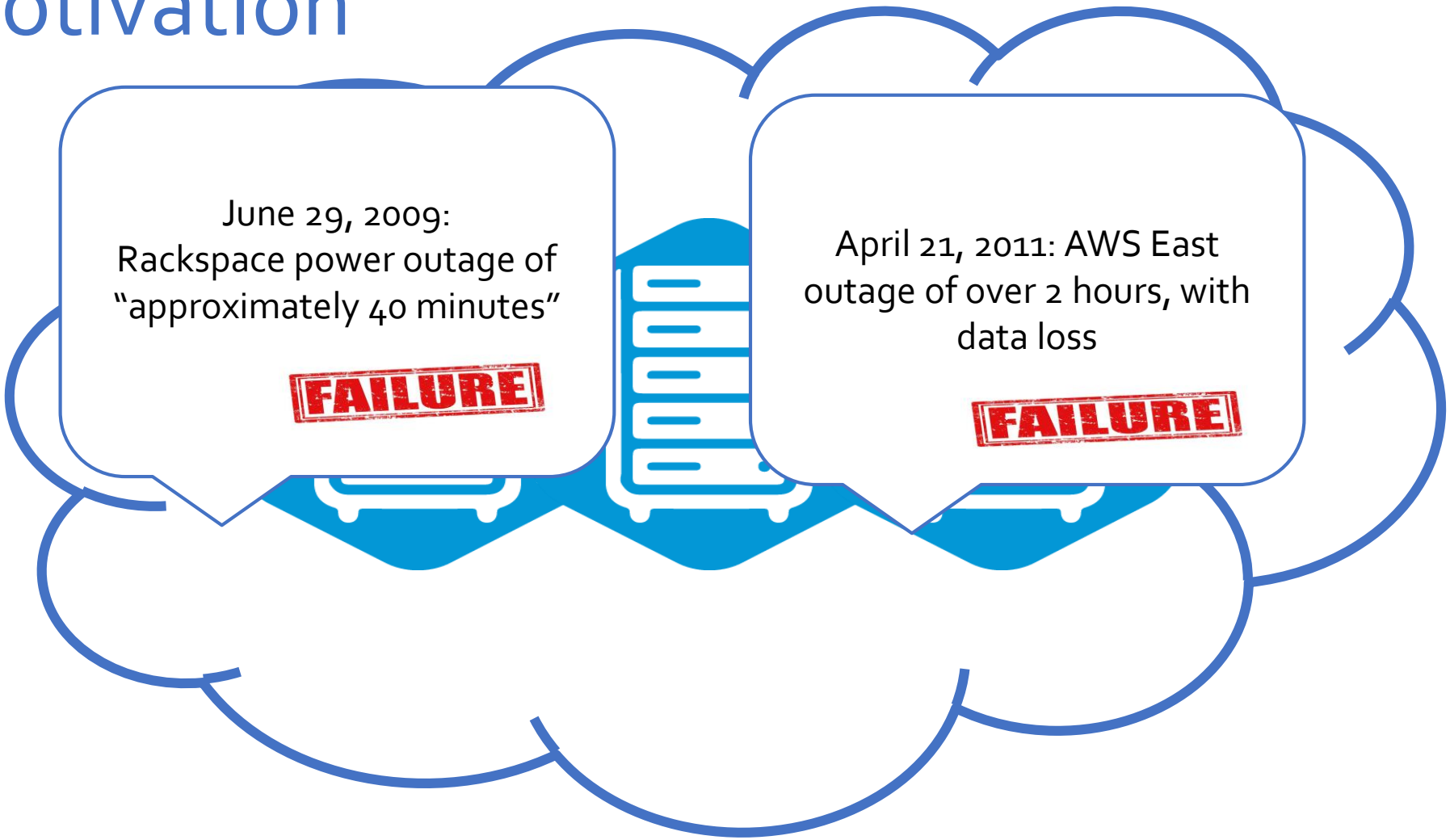
Presented By: Kartik Killawala

Acknowledgement

- Tim Kraska, Gene Pang, Michael Franklin : University of California, Berkeley
- Samuel Madden : Massachusetts Institute of Technology
- Alan Fekete : University of Sydney
- Published in EuroSys '13
- Some parts/slides of this presentation are from the official presentation of MDCC available on their website:

  http://mdcc.cs.berkeley.edu/mdcc_presentation_eurosys13.pdf

Motivation


June 29, 2009:
Rackspace power outage of
“approximately 40 minutes”
FAILURE
April 21, 2011: AWS East
outage of over 2 hours, with
data loss
FAILURE

Motivation

- Geo Replication to avoid the previously mentioned problems
- This causes High Network Latency
- Main Problems
  - High Network Latency
  - Unreliable Data Centers
- Challenge : To create protocol for a Fast and Reliable Database

Introduction

- Geo-replication is qualitatively different from replication within a cluster
- Delays are in hundreds of milliseconds (depending on the pair of locations) and could reach the limit of latency tolerance by a particular user
- Crucial to reduce the message round trips and desirable to avoid waiting for slowest data center to respond
- Traditional mechanism for transactions that are distributed is 2PC

Two-Phase Commit (2PC)

- Standard method for Distributed Transactions
- Two Phases : Prepare Phase + Commit Phase
- Serious Drawbacks in a Geo-replicated system
  - Depends on a Reliable Coordinator
  - Coordinator failure will block all the transactions currently executing
  - Transactions could have locks that prevent other Transactions to proceed until recovery is done

MDCC

- An optimistic commit protocol for geo-replicated transactions
- Single Wide Area message round trip
- It doesn't require a Master or Static partitioning
- It is strongly consistent at a cost similar to eventually consistent protocols
- MDCC takes advantage of Generalized Paxos for transaction processing and exploits commutative updates with value constraints in quorum based systems

Existing Solutions

	Consistency	Transactional Unit	Commit Latency	Data Loss Possible?
Amazon Dynamo	Eventual	None	1 round trip	Not Possible
Yahoo PNUTS	Timeline Per Key	Single Key	1 round trip	Possible
COPS	Causality	Multi Record	1 round trip	Possible
MySQL(async)	Serializable	Static Partition	1 round trip	Possible
Google Megastore	Serializable	Static Partition	2 round trips	Not Possible
Google Spanner	Snapshot Isolation	Partition	2 round trips	Not Possible
Walter	Parallel Snapshot Isolation	Multi Record	2 round trips	Not Possible

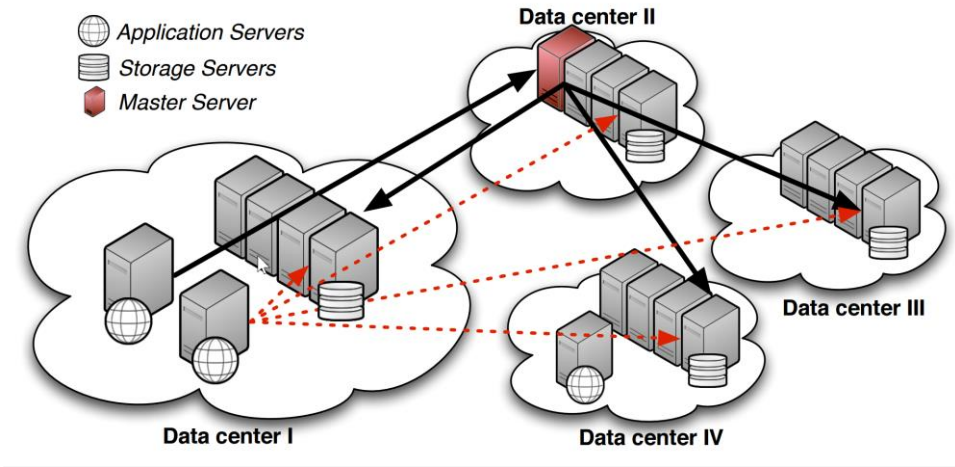
Existing Solutions + MDCC

	Consistency	Transactional Unit	Commit Latency	Data Loss Possible?
Amazon Dynamo	Eventual	None	1 round trip	Not Possible
Yahoo PNUTS	Timeline Per Key	Single Key	1 round trip	Possible
COPS	Causality	Multi Record	1 round trip	Possible
MySQL(async)	Serializable	Static Partition	1 round trip	Possible
Google Megastore	Serializable	Static Partition	2 round trips	Not Possible
Google Spanner	Snapshot Isolation	Partition	2 round trips	Not Possible
Walter	Parallel Snapshot Isolation	Multi Record	2 round trips	Not Possible
MDCC	Read Commited without Lost Updates	Multi Record	1 * round trip	Not Possible

MDCC

- Default Configuration – "Read Committed Isolation" without lost updates by detecting write-write conflicts
- Atomic Durability : Either all updates in a transaction eventually persist or none
- Read Committed : Updates from uncommitted transactions are never visible to other transactions
- No Lost Updates : Concurrent updates are either resolved (commutative) or prevented
- No Atomic Visibility : Some updates from committed transactions might be visible before all updates are visible

Architecture



- Uses library centric approach similar to architectures of Megastore and Spanner
- Every Data Center has full replica of data and data is partitioned across machines within a single data center
- Supports individual master per record
  - Storage Nodes can act on Masters behalf (black arrows)
  - Transaction Manager coordinates the update without acquiring mastership (red arrows)

MDCC Working Parameters

- Intra Data Center latencies are ignored
- Tradeoff between reducing latency by using more CPU cycle time to make sophisticated decisions
- Exploit 2 key observations of workloads
  1. Conflicts are Rare – Everyone updates their own data
  2. Many updates commute upto a limit

Paxos

- Paxos – A family of quorum based protocols for achieving consensus on a single value among a group of replicas
- Tolerates a variety of failures including lost, duplicated or reordered messages as well as failure and recovery of nodes
- Distinguishes between clients (*app servers*), proposers (*masters*), acceptors (*storage nodes*) and learners (*all nodes*)

Classic Paxos

- Every record has a master responsible for coordinating updates
- Operates in 2 Phases
- **Phase 1 –**
  - Master *P* selects a ballot number *m* and sends a *Phase1a* with *m* to majority of storage nodes responsible for the record *r*
  - Ballot number is unique for each master as it used to determine the latest request
  - If storage node receives a proposal number *m* greater than any previous responded messages, it responds with a *Phase1b* message with *m* , the highest numbered update alongwith its proposal number *n*
  - It also promises not to respond to any future requests with proposal number < *m*
  - If *P* receives responses from a majority of storage nodes, it has been chosen as master
  - Only *P* can commit a value for proposal number *m*

Classic Paxos

- **Phase 2**
  - *P* sends a *Phase2a* accept message to all storage nodes with *m* and value *v*
    - *v* is either the update of the highest numbered proposal among the *Phase1b* responses or the requested update from the client
  - Nodes accept the proposal unless they have responded to a *Phase1a* message of number > *m* and sends *Phase2b* message containing *m* and the value back to *P*
  - If *P* receives a majority of *Phase2b* messages with the same ballot number then a *consensus* is reached and the value is learned
- Single value per single instance is learned thus use single Paxos instance per version of the record assuming that the previous version has already been chosen successfully
- Drawbacks
  - Requires 2 message rounds

Multi Paxos

- It is an optimization over Classic Paxos in which if the master is reasonably stable, it is chosen as the master for several instances thus avoiding Phase 1 for several instances
- Meta Data *[Start Instance, End Instance, Ballot]*
- This also allows for different masters for different instances and the database stores this meta data including the current version as the part of the record which enables separate paxos instance per record
- For inserts, each table stores a default meta data value for any non existent records

MDCC - Transaction Support

- Extension of Multi Paxos to support multi record transactions
- Guarantee Atomic Durability, Detect Write-Write conflicts, Read committed
- Use Paxos instance per record to accept an *option* rather than writing the value directly
- After the app-server learns the *options* for all the records in a transaction, it commits and asynchronously notifies storage nodes to execute the *options*
- If *option* is not yet executed it is called *outstanding option*

Protocol

- As in all optimistic concurrency control techniques, they assume that the write set of all records is present at the end of the transaction which the protocol then tries to commit
- Updates v_{read} -> v_{write} this allows MDCC to detect write-write conflicts
- App server coordinates the transaction by trying to get the options accepted for all updates by proposing it to the Paxos instances for each record
- Storage nodes respond to the server with an accept or reject of the *option* depending on v_{read}

Protocol

- This is fundamentally different than the basic Paxos consensus based on the ballot number
- This change does not violate the Paxos assumption is because there was an assumption that a newer version can only be chosen if the previous version was successfully determined
- Thus all nodes will make the same commit or abort decision
- Similar to 2PC the app server learns an *option* if majority of storage nodes agree on that *option*
- In contrast to 2PC
  - MDCC doesnot allow clients or app servers to abort a transaction after being proposed
  - Decisions are determined and stored by the distributed nodes instead of only the coordinator

Protocol

- If app server determines that transaction is aborted or committed, it sends *Learned* message to the storage nodes
- Storage nodes execute the *option* or mark it as rejected
- At a time only one *option* per record can be outstanding as we require the previous instance
- If all record masters are local, it is possible to commit the transaction with a single round trip across the centers
  - Commit/Abort decision is depends on learned values
  - App server is not allowed to prematurely abort a transaction
- If master is not local, 2 round trips will be required

Deadlocks and Failure Scenarios

- Transaction might cause a deadlock by waiting on each others options
- Pessimistic strategy to avoid deadlocks – relax the condition that we can learn a newer version only if previous is committed
- Failure of Storage Nodes masked by use of quorums
- Master Failure can be recovered by selecting a new master using Phase 1 and 2 as described earlier
- App server failure is detected by simple timeouts, the state is reconstructed using a quorum of storage nodes for all keys in the transaction

Fast Paxos

- It avoids the master by distinguishing between classic and fast ballots
- Classic Ballots use the Classic Paxos algorithm
- Fast Ballots use quorum bigger than classic ballots but allow bypassing the master
- Since updates wont be serialized by the master, collisions may occur and have to be resolved using classic ballots
- MDCC uses fast ballots where all versions start implicitly using a fast ballot number unless changed by master by a *Phase1a* message
- This informs storage nodes to accept the next *options* from any proposer
- Simple Majority Quorums donot guarantee safeness of the protocol

Fast and Classic Quorums

- Any two quorums must have non empty intersection
- There is a non empty intersection between any three quorums where two are Fast Quorums and one is Classic Quorum
- If fast quorum can be achieved then value is safe and guaranteed to commit but if fast quorum is not achieved, collision recovery is necessary
- This collision is separate from transaction collision as this consists of nodes not agreeing on an option
- To resolve this, a new classic ballot is started

Fast Paxos

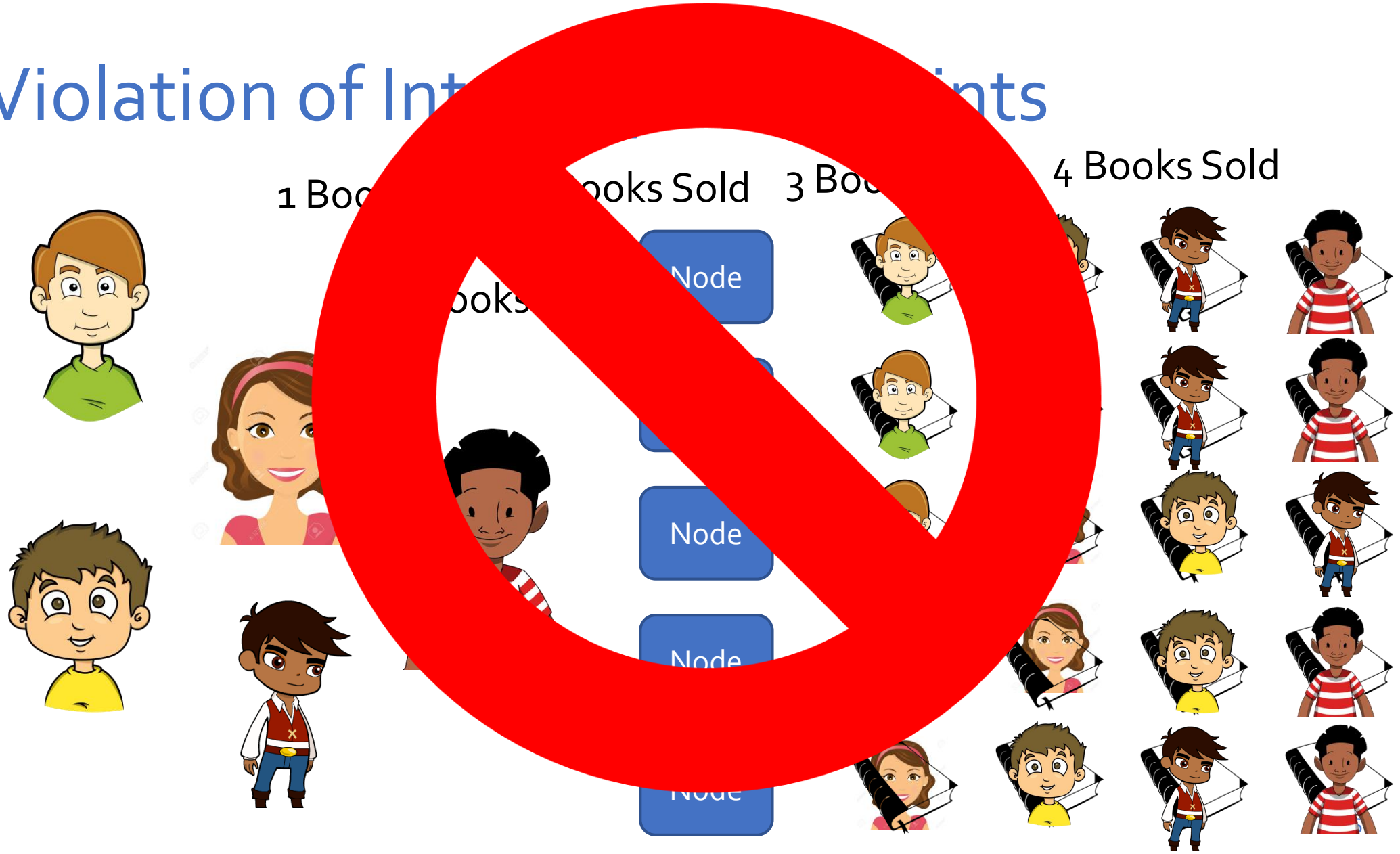
- Per fast ballot only one option can be learned but we can combine Fast Paxos with Multi Paxos using the following adjusted meta-data explanation *[Start Instance, End Instance, Fast, Ballot]*
- If collision occurs, instance is changed to classic and the collision is resolved and the protocol moves on
- Classic ballots have to be given priority over fast ballots
- Tradeoff Parameters
  - If collision occurs then 2 message rounds for resolution whereas classic only require 2 rounds
  - Thus fast paxos should only be used if conflicts and collision are rare
- MDCC assigns γ instances following a conflict to classic ballots (default 100)

Generalized Paxos

- Uses the same ideas as Fast Paxos but relaxes the constraint that acceptors must agree on the same exact sequence of values/commands
- As some commands can commute with each other they must agree on sets of commands compatible with each other

- This can sometimes lead to violation of integrity constraints that is very crucial in terms of database applications

Violation of Int nts

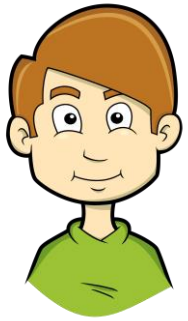


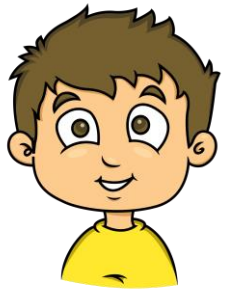
Solution

- Demarcation based strategy for quorum systems
- $L = \frac{N - Q_F * X}{N}$

Demarcation Limit

1 Book Sold





2Books Sold

Node

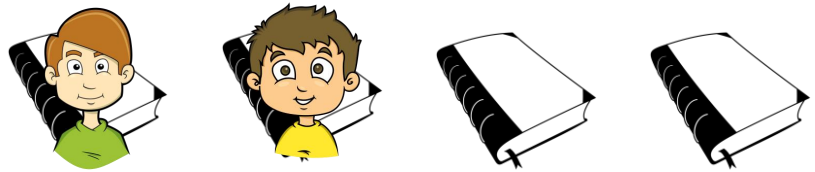
Node

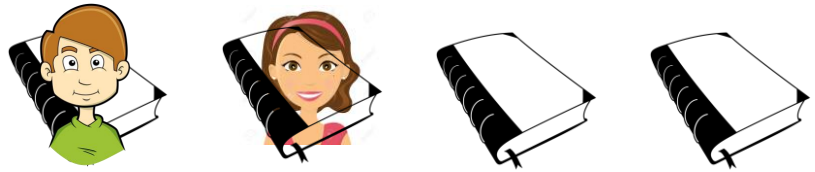
Node

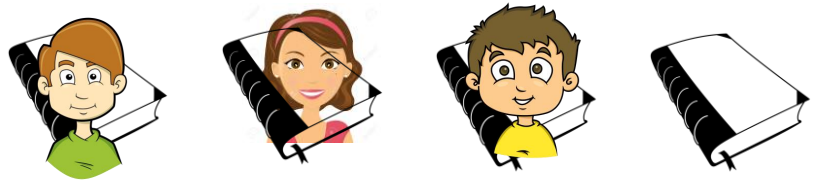
Node

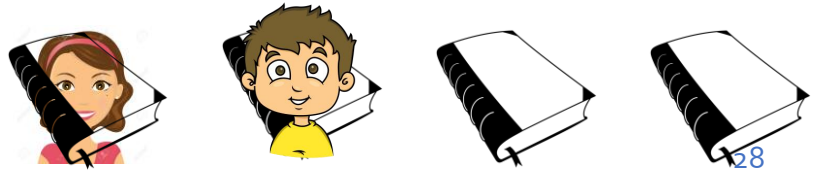
Node

3 Books Sold









Stock > (N-Q)/N * X
Stock > 0.8

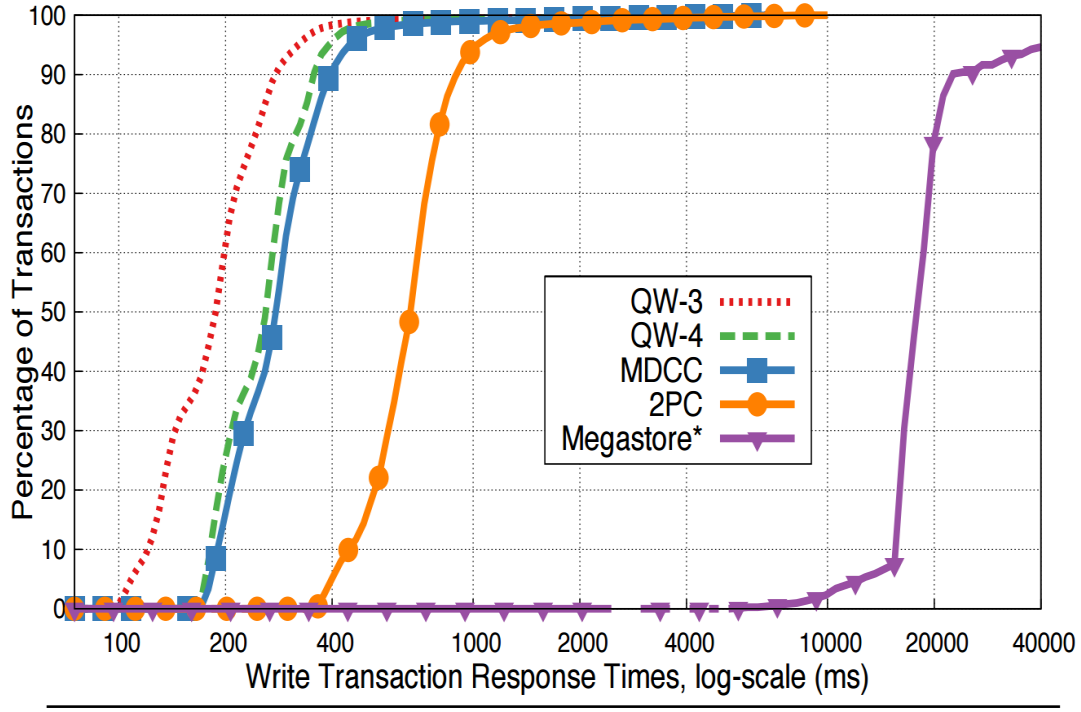
Consistency Guarantees

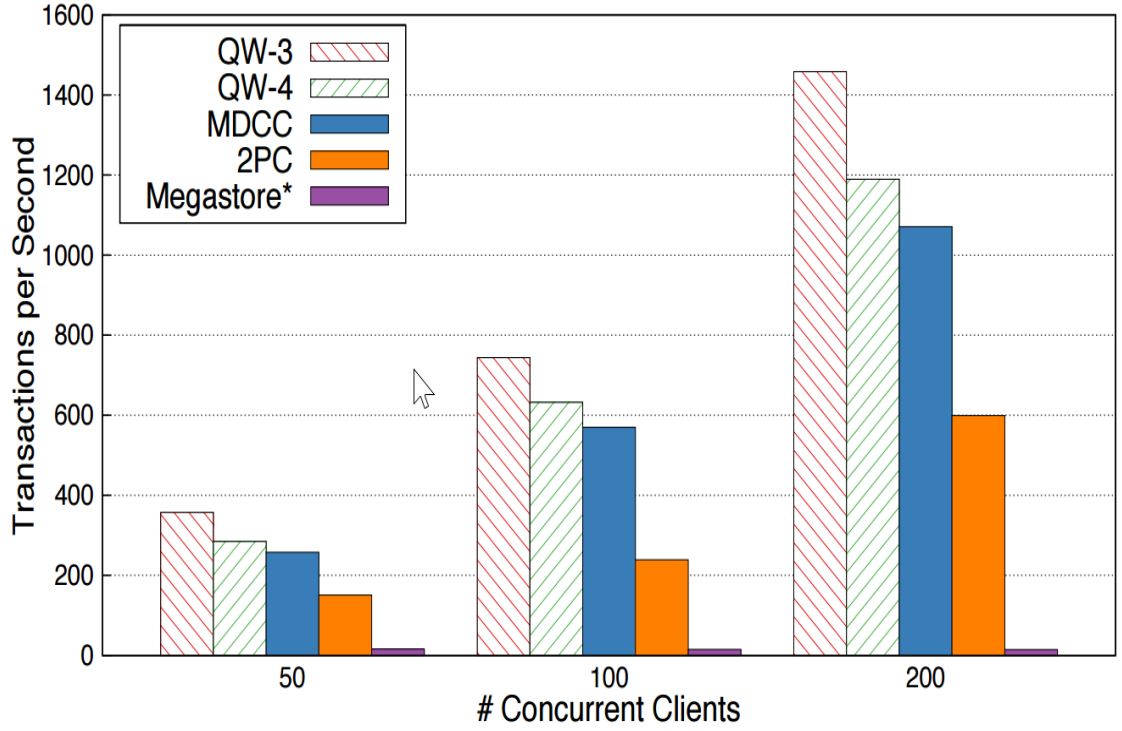
- Read Committed without Lost updates
- Staleness and Monotonicity
- Atomic Durability but No Atomic Visibility
  - Unless combined with other techniques such as 2 Phase Locking and Snapshot Isolation
- Other Isolation Levels
  - NMSI and Spanner's snapshot isolation are natural fits for MDCC
  - Protocol could be extended to consider read-sets allowing the leverage of optimistic concurrency control techniques and could provide full serializability

Evaluation

- Implemented on SCADS – a distributed key value store
- Five Data Centers using Amazon EC2 cloud (California, Virginia, Ireland, Singapore, Tokyo)
- TPC-W and Micro benchmarks used to compare and evaluate MDCC with different transactional and non transactional eventually consistent protocols

TPC-W Evaluation



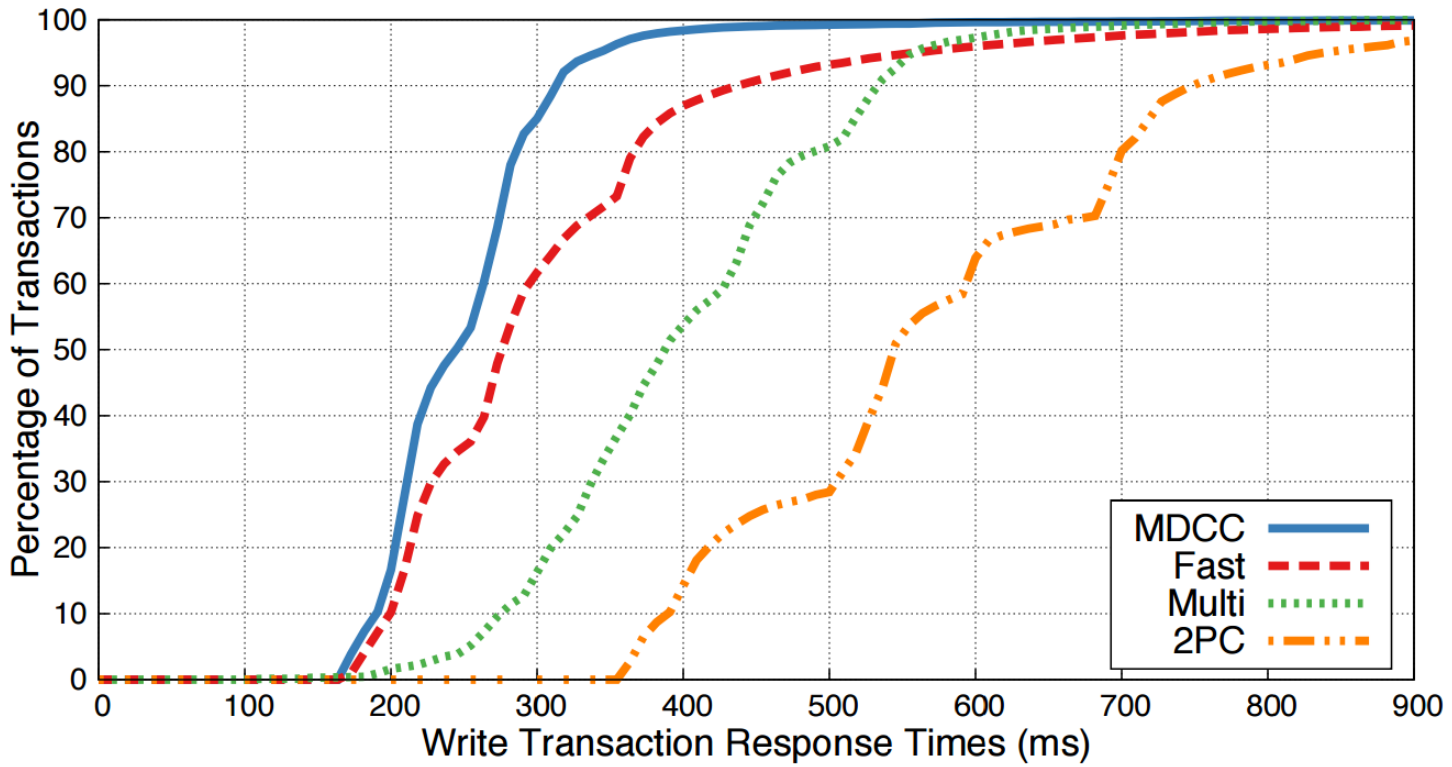


Micro-benchmark

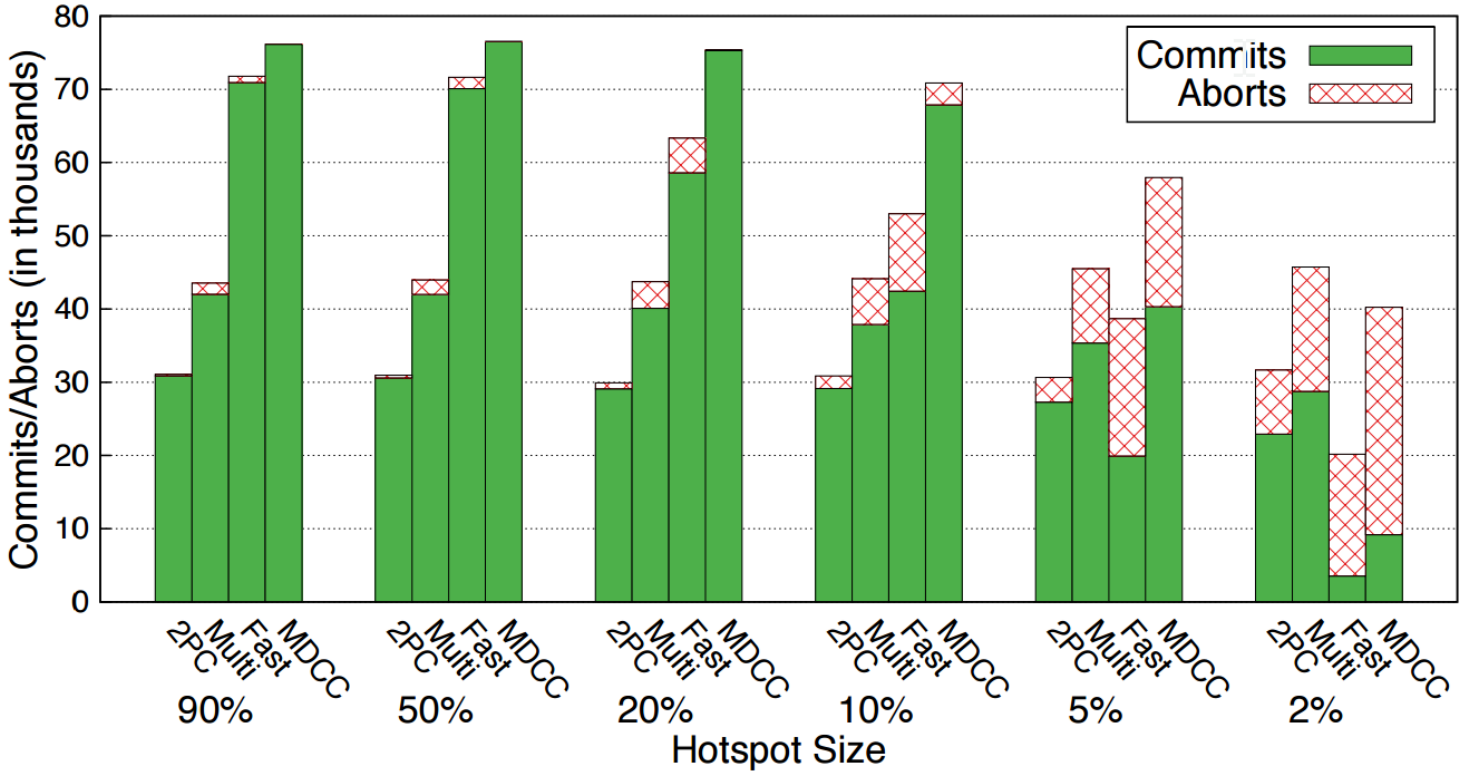
- Used their own micro benchmark to independently study the different optimizations within MDCC and compared it to the 2PC protocol

	Supports Multi Paxos	Supports Fast Paxos	Optimizes Commutative Updates
MDCC	✓	✓	✓
Fast	✓	✓	✗
Multi	✓	✗	✗

Write Response Time CDF



Varying Conflict Rates



Conclusion

- Conflicts are rare or Conflicts Commute
- MDCC is able to tolerate data center failures without compromising consistency at a similar cost to eventually consistent protocols
- It takes only 1* message round trip
- First technique presented to guarantee integrity constraints in quorum based systems