COMP90020: Distributed Algorithms

13. Atomic Commmit Protocols and Failure Recovery

Consistent Commits Are Hard in Distributed Transactions

Miquel Ramirez

THE UNIVERSITY OF MELBOURNE

Semester 1, 2019

Agenda

Agenda

1. Atomic Commit Protocols
2. Two-Phase & Three-Phase Commit
3. Failure Detection & Recovery
4. Biblio & Reading

Atomic Commit is a Fundamental Problem in DS

Atomic commitment problem (ACP) [Babaoglu & Toueg]

- Ensure a globally consistent transaction despite failures
- Decision is based on agreement among all participants
    - Commit: all participants make the transaction's update permanent
    - Abort: none will

Properties – Special Case of Consensus

- All participants that decide reach the same decision
- If any participant decides commit, then all participants must have voted "yes"
- If all participants vote yes and no failure occur, then all participants decide commit
- Each participant decides at most once (i.e. decision is not reversible)

The Atomic Commit Protocol (ACP) Problem

Conditions

Validity

- If a coordinator broadcasts a message m, then all participants eventually receive m

Integrity

- For any message m, each participant receives m at most once and only if a coordinator actually broadcasts m

Timeliness (only for synchronous systems)

- There is a known constant d (delay) such that a broadcast of m initiated at time t, is received by every participant by $t + d$

One-Phase Commit Protocol

Reminder

Transactions come to an end when client requests commit or abort.

One-Phase Commit (1PC) Protocol

1. Coordinator receives request from client
2. Coordinator broadcasts message indicating commit or abort
3. When all participants have sent back acknowledgment exit, otherwise broadcast instruction again

One-Phase Commit Protocol

Reminder

Transactions come to an end when client requests commit or abort.

One-Phase Commit (1PC) Protocol

1. Coordinator receives request from client
2. Coordinator broadcasts message indicating commit or abort
3. When all participants have sent back acknowledgment exit, otherwise broadcast instruction again

(Mostly) Crippling Limitation of 1PC

Coordinator cannot decide to abort when client requests commit

- Concurrency control requires abort (deadlock, failed validation)
- Server may have crashed and been restarted (which conveys to abort the transaction)

Agenda

Failure Model

Process Crashes

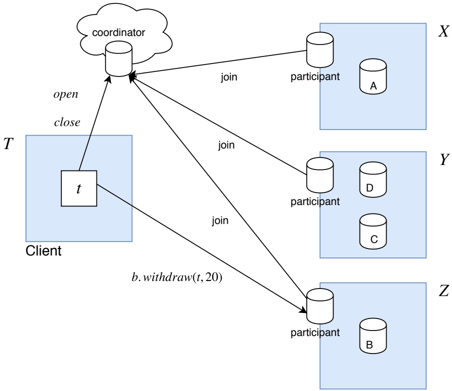
- Any process can crash at any time.
- This includes the coordinator of the transaction.

Bounded Delays in Message Delivery

- Messages are delayed up to d time units
- A message sent at time t will be received some time before $t + d$.

Two-phase Commit Protocol (2PC) - Starting State

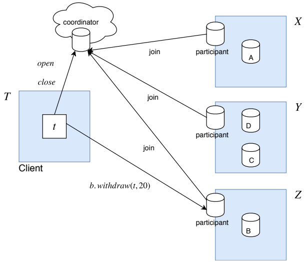
Servers involved in transaction t



- Coordinator has list of all participants (a.k.a. *cohorts*).
- Protocol kickstarted by client invoking $close()$ on the coordinator.

2PC - Voting Phase Overview

1. Coordinator decides whether to commit or not.
2. If decision is to commit, send *request* to participants, who cast votes
3. on the basis of votes, coordinator makes final decision.



2PC follows a unanimous voting scheme:

- The transaction will commit only if all participants vote to do so.

Question!

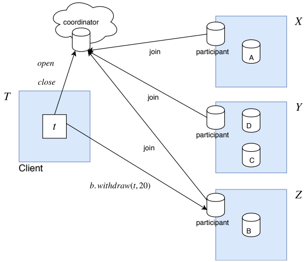
Shouldn't be the client a participant too?

(A): Yes

(B): No

2PC - Voting Phase Overview

1. Coordinator decides whether to commit or not.
2. If decision is to commit, send *request* to participants, who cast votes
3. on the basis of votes, coordinator makes final decision.



2PC follows a unanimous voting scheme:

- The transaction will commit only if all participants vote to do so.

Question!

Shouldn't be the client a participant too?

(A): Yes (B): No

→ (No): The client is the initiator.

Reason for Aborts

Question!

Assume that we use locking (correctly) for concurrency control and we have taken care to ensure all schedules are ACA. Which of the following are possible reasons for a participant to vote against committing the transaction or the coordinator to decide $abort$**?**

(A): A deadlock

(B): Dirty reads

(C): Dirty writes

(D): A server crashed

Reason for Aborts

Question!

Assume that we use locking (correctly) for concurrency control and we have taken care to ensure all schedules are ACA. Which of the following are possible reasons for a participant to vote against committing the transaction or the coordinator to decide $abort$**?**

(A): A deadlock

(B): Dirty reads

(C): Dirty writes

(D): A server crashed

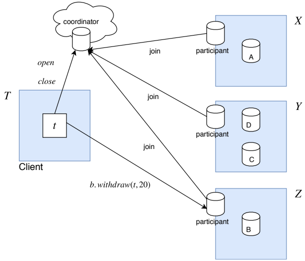
→ (A & D): Dirty reads and dirty writes are not possible. Deadlocks can happen since we have no control over the code in the transactions being executed concurrently. Crashes can happen too.

2PC - Completion Phase Overview

Servers assumed to have three kinds of storage: volatile (RAM), stable (private hard disk), permanent (public hard disk with long term backup).

Tentative Versions

Each participating server keeps track of updates it makes on the objects served, keeping the initial, and the posterior sequence of modified values.



If commit is the decision reached

- Tentative versions of shared objects are written to permanent storage

Otherwise, these changes are discarded.

Question!

Question!

In the view of the property of isolation should tentative versions be visible to other servers or transactions?

(A): Yes (B): No

Question!

Question!

In the view of the property of isolation should tentative versions be visible to other servers or transactions?

(A): Yes

(B): No

→ (No!): Tentative versions, as is the case when we use Optimistic Concurrency Control, must remain invisible to other remote processes or local threads, before a decision to commit is made.

Operations and Messages - I

$canCommit(t)$: Coordinator → Participant

- Message sent by coordinator to request a vote
- Participant responds with either **Yes** or **No**

$doCommit(t)$: Coordinator → Participant

- Coordinator instructs participant to commit

$doAbort(t)$: Coordinator → Participant

- Coordinator instructs participant to rollback objects to initial version

Operations and Messages - II

$haveCommitted(t, p)$: Participant →Coordinator

- Participant asks coordinator to confirm transaction t is committing

$getDecision(t)$: Participant →Coordinator

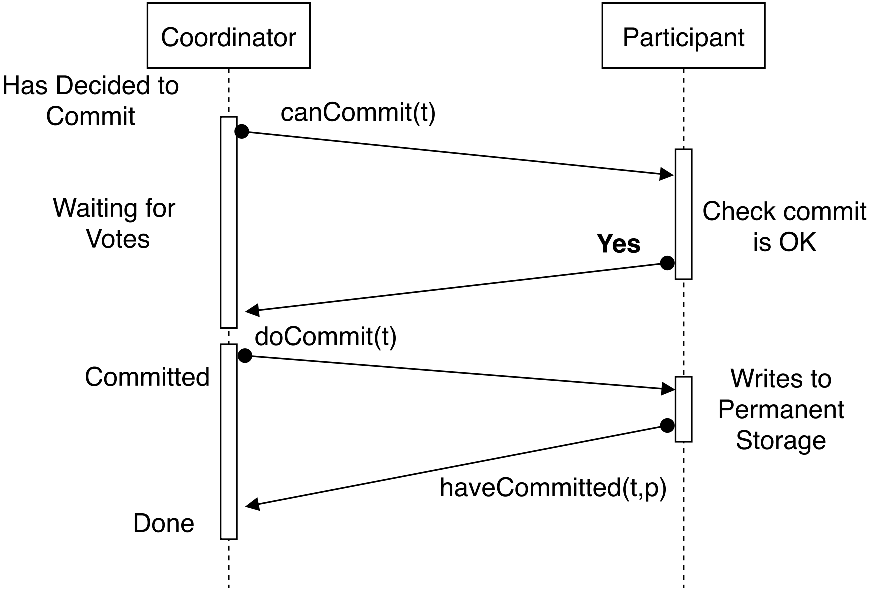
- Asks for the decision on transaction t,

Notes on $getDecision(t)$:

- the participant sending $getDecision(t)$ voted **Yes**,
- but has not received $doCommit(t)$ or $doAbort(t)$ after a fixed amount of time.

→ Purpose: Used to recover from server crash or delayed messages.

2PC - Message Exchange Chronogram



Voting Phase – Message Exchange & Control Flow

Protocol for Voting Phase

1. Coordinator broadcasts message $canCommit(t)$ to participants (and itself)
2. Recipients reply

   Yes If operations on hosted shared objects successful

   No If some conflict occurred or a local deadlock is detected.

Question!

If a server crashes right after replying Yes, can we avoid to abort the transaction?

(A): No

(B): Saving tentative values and restarting?

Voting Phase – Message Exchange & Control Flow

Protocol for Voting Phase

1. Coordinator broadcasts message $canCommit(t)$ to participants (and itself)
2. Recipients reply
   - Yes If operations on hosted shared objects successful
   - No If some conflict occurred or a local deadlock is detected.

Question!

If a server crashes right after replying Yes, can we avoid to abort the transaction?

(A): No

(B): Saving tentative values and restarting?

→ (B): once it is known that the transaction needs to commit, we can write tentative vlaues to stable storage. They can then be used as a recovery point for when the server restarts.

Completion Phase – Message Exchange & Control Flow

Protocol for Completion Phase

1. Coordinator counts votes, if:
   - → All Yes: coordinator broadcasts $doCommit(t)$
   - → At least one No: coordinator broadcasts $doAbort(t)$
2. If participant receives
   - $doCommit(t)$
     a. tentative versions permanently stored,
     b. sends $haveCommitted(t, p)$ to coordinator to acknowledge the received command.
   - $doAbort(t)$
     - rollback to initial values,
3. If transaction commits, Coordinator returns control to client when all $haveCommitted(t, p)$ received from every server.

2PC Protocol - Coping with Crashes

Failure Detector

Coordinator and participants have as a service a complete and reliable failure detector:

- keeps set $H(p, t)$ of processes suspected to have crashed at time t,
- each server needs to send hearbeat message to coordinator on a regular basis.

Properties of failure detectors:

- reliable $\rightarrow$ only suspects of crashed processes,
- complete $\rightarrow$ there are no false positives

2PC Protocol - Handling Failures

Voting Phase

- If participant crashes,
  - Coordinator counts that as No vote to $canCommit(t)$.

Completion Phase

- If participant crashes after receiving $doCommit(t)$,
  - tentative values must be made visible,
  - we can guarantee this by writing them to stable storage and restarting.

2PC protocol - Where it All Goes Wrong

2PC Flaw - Does Not Guarantee Agreement

Suppose that all of the following happens

- Coordinator crashes at the start of the completion phase.
- A participant crashes slightly later
- Coordinator sent $doCommit(t)$ or $doAbort(t)$ to crashed participant only,
- and made tentative values visible, or undid these irrevocably.

2PC protocol - Where it All Goes Wrong

2PC Flaw - Does Not Guarantee Agreement

Suppose that all of the following happens

- Coordinator crashes at the start of the completion phase.
- A participant crashes slightly later
- Coordinator sent $doCommit(t)$ or $doAbort(t)$ to crashed participant only,
- and made tentative values visible, or undid these irrevocably.

Failure Handling

- put in place fresh coordinator, who reassesses transaction OR,
- participants now must decide whether transaction commits or aborts,
- meanwhile, participants are indefinitely blocked.

Alternative: make coordinators reliable via replication (e.g. Google's Cloud Spanner/Paxos, or FaunaDB/RAFT)

2PC in Perspective

Used in sharded DBs when transaction uses data on multiple *shards* (servers).

Several outstanding issues preclude wider usage

- Slow due to messaging and requiring many writes to disk,
- locks held for long times, reduces throughput
- coordinator crashes a fatal error, leading to deadlocks

Faster distributed transactions are active research area:

- Lower message and persistence cost
- Identify special cases that can be handled with less work
- Allow wide-area transactions (between banks, airlines etc.)
- Less consistency guarantees, more burden on applications

Three-Phase Commit (3PC) protocol

Idea To Guarantee Agreement

Coordinator enters an intermediate precommit phase after receiving Yes from all participants.

$\rightarrow$ TL;DR: Add an extra round

Two new messages

$preCommit(t)$: Coordinator $\rightarrow$ Participant

- Coordinator tells participant it is prepared to commit

$ackPreCommit(t, p)$: Participant $\rightarrow$ Coordinator

- Acknowledges receipt of $preCommit(t)$

3PC - Additional Steps

1. 2PC Voting phase
2. Coordinator broadcasts $preCommit(t)$ to participants,
3. upon receiving $preCommit(t)$, each participant replicies with $ackPreCommit(t)$,
4. after receiving $ackPreCommit(t)$ from every particpant, coordinator broadcasts $doCommit(t)$,
5. 2PC Completion phase

3PC Failure Handling

If participant crashes before sending $ackPrecommit(t)$

- coordinator *may* still decide to commit transaction,
- if so, process replacing crashed participant then completes its part of the transaction.

3PC Failure Handling

If participant crashes before sending $ackPrecommit(t)$

- coordinator *may* still decide to commit transaction,
- if so, process replacing crashed participant then completes its part of the transaction.

If coordinator crashes in the precommit phase,

- participants can agree among themselves whether to commit the transaction,
- since they didn't roll back or make visible their tentative versions.

Is 3PC a Winner?

Question!

3PC is not used very often, can you think of the reasons for that being the case?

(A): People don't read books

(B): Less efficient than 2PC

(C): Network reliability

(D): None of these reasons

Is 3PC a Winner?

Question!

3PC is not used very often, can you think of the reasons for that being the case?

- (A): People don't read books
- (B): Less efficient than 2PC
- (C): Network reliability
- (D): None of these reasons

→ (B & C): 3PC only works if network reliable, or if participants have reliable failure detector that can tell apart the coordinator being dead from the network not delivering packets. For example, 3PC won't work correctly if there's a network partition. In most practical networks, partition is possible.

Agenda

Coping with Crashes

"Phoenixing" via checkpointing and rollback recovery

→ When a process crashes, another (existing or new) process resumes its execution from saved state and message log.

Assumptions

- The network topology is complete and communications reliable.
- Failure detector that is complete and reliable available to processes.

Requirements

Each process p stores part of its local state into stable storage, that remains accessible to the other processes in a consistent state after p has crashed.

Factoids - Checkpoints are not Snapshots

#1: Each process periodically saves its state and received messages in stable storage

- No synchronization with other processes, crashes assumed to be rare.

#2: When the process crashes, alive processes then perform a rollback towards a consistent configuration, and restart execution.

#3: An event (change in the state of the proces) is rolled back if

- it happened after the last checkpoint at the crashed process,
- or has causal dependency with respect to rolled back events.

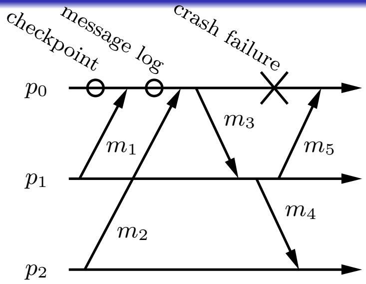
#4: Stable storage can be implemented using two disks (RAID)

- wallet alert!
- Memory updates in first disk are faithfully copied to the second disk.
- Checkpointing and message logging is performed sporadically.

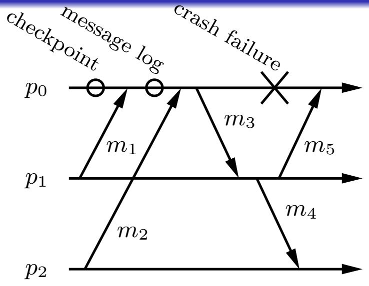
Rollback Recovery

When p_0 recovers from its crash, state is restored to its last checkpoint, and the receipt of m_1 is replayed.

It is like p_0 never received m_2, that fact is lost. This is an inconsistent state (cut) as m_2 is lost and m_3 is "orphan".



Rollback Recovery



When p_0 recovers from its crash, state is restored to its last checkpoint, and the receipt of m_1 is replayed.

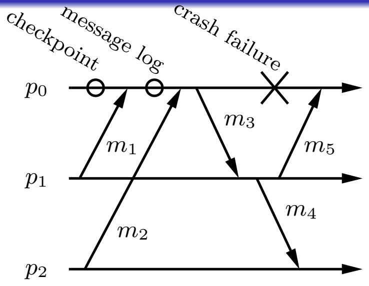
It is like p_0 never received m_2, that fact is lost. This is an inconsistent state (cut) as m_2 is lost and m_3 is "orphan".

p_1 needs to be rolled back to before the receipt of m_3, as it was sent by p_0 as a result of receiving m_2. This turns m_4 and m_5 into "orphans" too.

Then p_2 is in turn rolled back to before the receipt of m_4 and resends m_2.

Rollback Recovery

When p_0 recovers from its crash, state is restored to its last checkpoint, and the receipt of m_1 is replayed.



It is like p_0 never received m_2, that fact is lost. This is an inconsistent state (cut) as m_2 is lost and m_3 is "orphan".

p_1 needs to be rolled back to before the receipt of m_3, as it was sent by p_0 as a result of receiving m_2. This turns m_4 and m_5 into "orphans" too.

Then p_2 is in turn rolled back to before the receipt of m_4 and resends m_2.

Danger!: When m_5 arrives, p_0 needs to recognize it is an orphan message and discard it.

Peterson-Kearns Rollback Recovery Algorithm

Overview

- A vector clock is used to determine which events should be discarded in the rollback.
- During the rollback procedure, the computation being performed is stalled.
- The algorithm cannot cope with multiple concurrent crashes.
- Ignores possibility of crashes during recovery phase
  - → this is reasonable as long as crashes rare and recovery fast

Peterson-Kearns algorithm - Vector clock

Lamport's Logical Clocks

Each message contains the logical time of its send event, so that the logical time of the corresponding receive event can be determined.

The logical time of a process is the time of its last event

- initially it is $[0, \ldots, 0]$

As each checkpoint, both states and the process logical time saved in stable storage.

Vector times of incoming messages are kept in the message log.

Question!

Can we reconstruct exactly the vector clocks with the information above and message replay?

(A): Always

(B): Sometimes

(C): Never

(D): Often

Peterson-Kearns Algorithm - Restart Protocol

When crashed process p_i restarts

- last checkpoint and message log is retrieved from stable storage.

From the checkpoint, p_i replays events until message log exhausted.

Then p_i broadcasts message indicating number of events processed, k_i

- If last event replayed has vector time $[k_0, \ldots, k_i, \ldots, k_{N-1}]$,
- then it sends pair of numbers (k_i, i).

This initiates a rollback procedure at the other processes, so events with vector time's ith coordinate $> k_i$ are discarded.

Peterson-Kearns algorithm - Rollback Protocol

When process q receives (k_i, i), it checks whether the ith coordinate of its vector time is $> k_i$.

If so, q restarts at its last checkpoint for which the vector time's ith coordinate is $\leq k_i$.

It replays events up to (but not including) the first event for which the vector time's ith coordinate is $> k_i$.

Peterson-Kearns algorithm - Rollback Protocol

When process q receives (k_i, i), it checks whether the ith coordinate of its vector time is $> k_i$.

If so, q restarts at its last checkpoint for which the vector time's ith coordinate is $\leq k_i$.

It replays events up to (but not including) the first event for which the vector time's ith coordinate is $> k_i$.

Messages received by q beyond this point are kept only if the ith coordinate of their vector time (from corresponding send event) is $\leq k_i$.

These are clustered after the point where the replay at q halted.

Peterson-Kearns algorithm - Sequence numbers

Ghosts from Christmas Past

An "orphan" message, for which the corresponding send event was rolled back, may arrive after completion of the recovery phase.

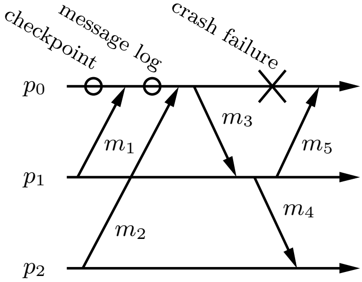
To ensure that orphan messages are recognized and discarded

1. Each process p has a sequence number seq_p, which initially is 0, and is increased by 1 at each new recovery phase.
2. seq_p, paired with the time stamp (k_i, i) of the corresponding recovery phase, is kept in stable storage.
3. seq_p is attached to each message sent by p.

Peterson-Kearns algorithm - Example

The sequence number initially is 0.

All messages carry this number.

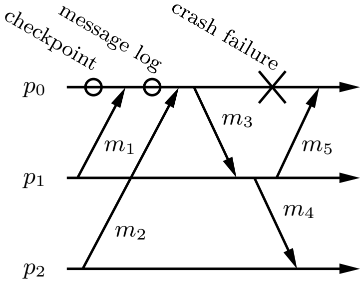


Peterson-Kearns algorithm - Example

The sequence number initially is 0.

All messages carry this number.

p_0 restarts from its last checkpoint with $seq_{p_0} = 1$ and replays the receipt of m_1 from its message log, with time $[k_0, k_1, k_2]$.



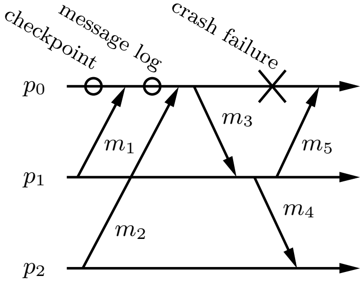
p_0 sends $(k_0, 0)$ to p_1 and p_2. These messages are received, and p_1 and p_2 start the rollback procedure with sequence number 1.

Peterson-Kearns algorithm - Example

The sequence number initially is 0.

All messages carry this number.

p_0 restarts from its last checkpoint with $seq_{p_0} = 1$ and replays the receipt of m_1 from its message log, with time $[k_0, k_1, k_2]$.



p_0 sends $(k_0, 0)$ to p_1 and p_2. These messages are received, and p_1 and p_2 start the rollback procedure with sequence number 1.

By m_3 and m_4, the vector times at p_1 and p_2 are $> k_0$ at index 0.

So they restart at their last checkpoint (not shown in the picture) and replay events, until right before the receipt of m_3 and m_4.

Peterson-Kearns Algorithm - Resending Messages

Rule is to Always Resend to Crashed Processes

Send events to the *crashed* process p_i that weren't rolled back are repeated by the sender.

Question!

This sounds wasteful, why do we need to do this?

(A): Receive event lost | (B): Just in case

Peterson-Kearns Algorithm - Resending Messages

Rule is to Always Resend to Crashed Processes

Send events to the *crashed* process p_i that weren't rolled back are repeated by the sender.

Question!

This sounds wasteful, why do we need to do this?

(A): Receive event lost

(B): Just in case

→ Because the corresponding receive event may have been irrecoverably lost in the crash.

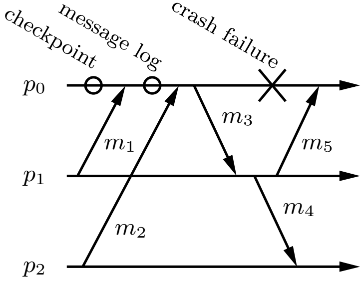
If not lost, p_i will recognize from the vector time of the send event that the message was already received and can discard it.

Peterson-Kearns algorithm - Example (continued)

m_1 and m_2 are resent by p_1 and p_2.

At p_0, m_1 is discarded because it is in p_0's message log;

But m_2 is treated as a new message.

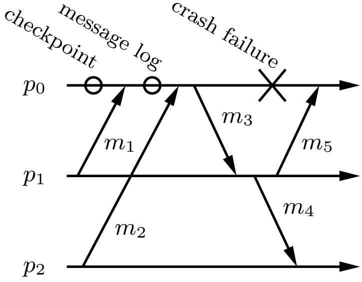


Peterson-Kearns algorithm - Example (continued)

m_1 and m_2 are resent by p_1 and p_2.

At p_0, m_1 is discarded because it is in p_0's message log;

But m_2 is treated as a new message.



When m_5 reaches p_0, it is discarded, because its sequence number is 0, and the vector time of its send event carries a value $> k_0$ at index 0.

Agenda

Further Reading

Coulouris et al. *Distributed Systems: Concepts & Design*

- Chapter 17, Section 3, Atomic Commit Protocols

Wan Fokkink's *Distributed Algorithms: An Intuitive Approach*

- Chapter 3, Section 3, Peterson-Kearns Rollback Algorithm

Refer to Kulik's coverage of

- Vector logical clocks
- Failure detectors

Spanner vs. Calvin

`https://blog.yugabyte.com/google-spanner-vs-calvin-global-consistency-at-scale/`

CAUTION: good info but author wants to promote his own stuff.