Coordinating Distributed Systems

Theory and practice

Daniele Venzano

Eurecom

Introduction

Outline

- **What is a distributed system?**
- **The consensus problem**
  - A few examples of distributed consensus
  - CAP theorem
  - Eventually consistent Vs Strongly consistent
  - Fault tolerance: possible faults in distributed systems
- **Consensus protocols**
  - Two phase commit
  - Paxos overview
  - Raft from A to Z
- **Implementations - ZooKeeper**
  - History
  - Architecture
  - Data model
  - Higher-level primitives

What is a distributed system?

- A set of processes seeking to achieve some common goal by communicating with each other

What is a distributed system?

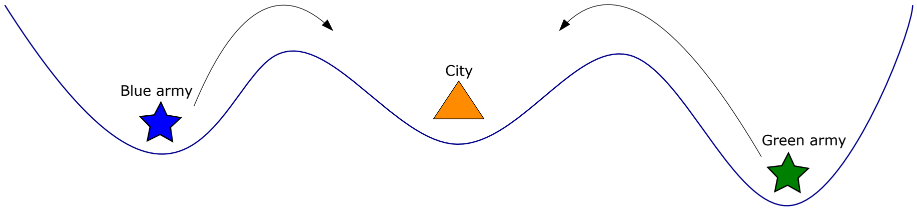
- Like a software matrioshka
  - Multi-threaded process
  - Multi-process on a single server
  - **Multiple processes in a set of servers in the same datacenter**
  - **Multiple processes in a set of geographically distributed servers**
- Why?
  - Inherent distribution (sensors, peer-to-peer, publish-subscribe, ...)
  - Engineering choice (fault tolerance, replication, performance, ...)
- These processes need to coordinate to reach a common goal:
  - data aggregation
  - synchronization
  - transactions
  - ...

The consensus problem

Wedding consensus

- The priest follows a well known protocol to reach a consensus:
  1. Priest: Alice, will you marry Bob ?
  2. Alice: yes
  3. Priest: Bob, will you marry Alice ?
  4. Bob: yes
  5. Priest: You are now husband and wife
- In distributed systems this becomes:
  1. Coordinator: Alice, can you commit key X with value 5 ?
  2. Alice: yes, I can
  3. Coordinator: Bob, can you commit key X with value 5 ?
  4. Bob: yes, I can
  5. Coordinator: Ok, both of you record that X has now a value of 5
- What if Bob flees from the church?

The two generals



- Two generals want to attack a city
- They can only use unreliable messengers to communicate
- They need to attack at the same time to succeed

An infinite number of messages is needed for each general to be sure the other agrees on the time of the attack.

A few examples

Common functionality in distributed systems:

- aggregate functions: sensors calculating average temperature
- synchronization: agree on a value
- reliable broadcast: a message sent to a group is received by all or none
- atomic commit: ensure that processes reach a common decision whether to commit or abort a transaction
- leader election: ensure there is only one process in charge at a given time

The CAP theorem

Properties of a distributed system

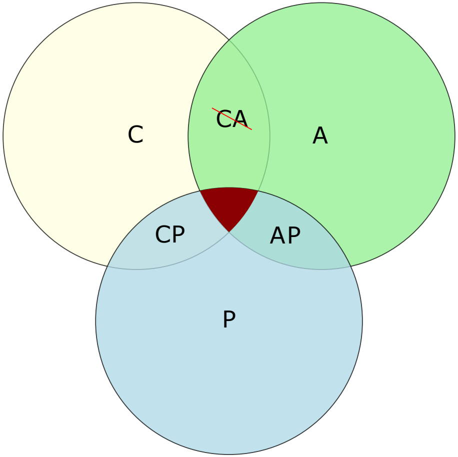
- Linearizability[1]: a given set of operations is *linearizable* if it appears to the rest of the system to occur instantaneously
  - Writes are linearizable operations if every read receives the most recent write
- Availability: a system is *available* if every request to a non-failing node always receives a response, eventually
  - Reading stale data is ok, though
- Partition tolerance
  - The system continues to function properly even if the network loses or delays an arbitrary number of messages

The CAP theorem links these three properties.

[1] The CAP theorem calls Consistency what is actually Linearizability

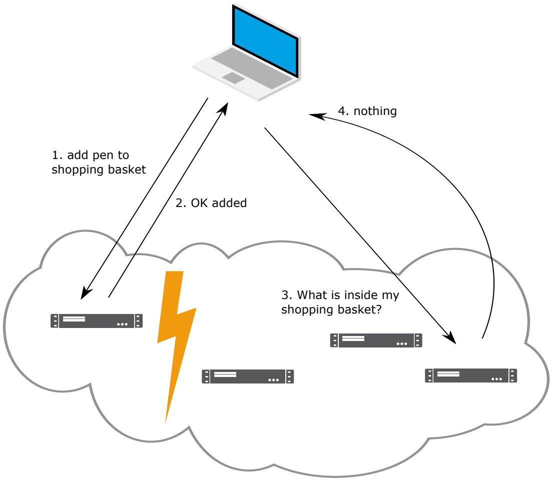
CAP theorem - 1

The theorem says: between C A P, you can choose only two



(... but you need the P ...)

CAP theorem - 2 - Why not all three?


1. add pen to shopping basket
2. OK added
3. What is inside my shopping basket?
4. nothing

If there is a network partition either C or A will break

CAP theorem - 3 - P

- Network partitions occur outside anyone's control in real life
- Cannot sacrifice the Partition-Tolerance property
- In the event of a network partition either A or C is maintained: it is the choice of the designer
- Practical distributed systems are CP or AP
- Some can be configured to shift between CP and AP (tunable consistency)

Note: a network partition can also be a very slow link

CAP theorem - 4 - Summary

- First stated by Eric Brewer (Berkeley) at the PODC 2000 keynote
- Formally proved by Gilbert and Lynch, 2002[4]

- The CAP theorem formally states the trade-offs among different distributed systems properties
- In practice network Partitions occur, the designer can choose one of Consistency or Availability
- The choice heavily depends on what your application/business logic is

CAP theorem - 5 - Summary

CP-oriented systems:

- BigTable, Hbase, MongoDB, Redis, MemCacheDB, Scalaris, Paxos, ZooKeeper[1]

AP-oriented systems:

- Amazon Dynamo, CouchDB, Cassandra[2], SimpleDB, Riak, Voldemort

In-depth articles on the misuse of the CAP theorem in describing real systems:

- `https://codahale.com/you-cant-sacrifice-partition-tolerance/`
- `https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html`

[2]CA or CP tunable

Consistency models

- Eventual consistency: after a successful write, every read from the distributed system will *eventually* return the written value, if no new updates are made to it
- Strong consistency: after a successful write, all reads from the distributed system will return the new value

Example: update a tag on a Facebook photo (a geo-distributed database)

Fault tolerance in distributed systems

Fault examples:

- Simple: network partitions, hardware or software crashes, outdated/malicious nodes (bizantine faults)
- Complex: feedback loops that overcompensate in good faith (examples in next slides)

Faults will always happen, design tolerance mechanisms:

- Replication: master-slave ($\rightarrow$ failover) or load balancing
- Isolation: malfunctioning components do not affect the system as a whole

Amazon DynamoDB crash (2015/09/20)

DynamoDB: distributed NoSQL database from Amazon web services (AWS)

1. Network disruption caused timeouts of some storage servers
2. The affected servers tried to re-establish their membership
3. A change of usage pattern caused membership data to be unexpectedly large
4. Membership servers started to overload
5. More storage servers timed-out on health checks to membership servers
6. Cascading failure, stabilized at 55% error rate for customers

Post-mortem: `https://aws.amazon.com/message/5467D2/`

More real-world failures

- Google: `https://status.cloud.google.com/summary`
- Facebook: `https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919`
- Apple: `http://appleinsider.com/articles/16/06/02/apples-app-stores-apple-tv-itunes-other-services-h`
- Microsoft (Azure): `https://azure.microsoft.com/en-us/status/history/`

Consensus protocols

Consensus protocols overview

Simplest (but unsafe):

- Two-phase commit

Traditionally studied for modern distributed systems:

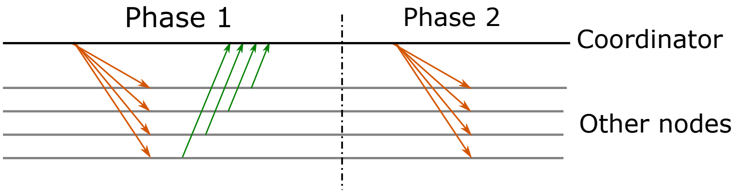
- Paxos
- Raft
- ZAB

Other:

- Lock-step (used in some multiplayer video games)
- The proof-of-work from Bitcoin

Two-phase commit

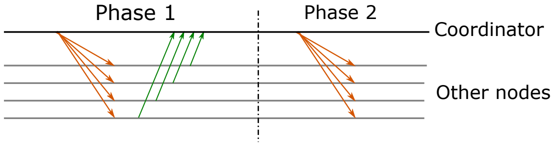
Simplest protocol for consensus



- Phase 1
  - One coordinator node *C* suggests a value to the other nodes
  - C gathers the responses
- Phase 2
  - If all nodes agree, C sends a commit command, abort otherwise

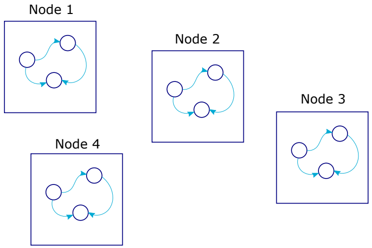
Two-phase commit - Why it is not enough?

Two phase commit cannot make progress if there are simple failures.[10]



- If C fails, nothing can be done until it is restarted
  - If C fails in phase 1, it has to abort the commit and retry
  - If C fails in phase 2, it has to replay the outcome (commit/abort)
  - In both cases the outcome is uncertain until C restarts
- If just one of the other nodes crashes, is slow or unreachable, the value cannot be committed.

State-machine replication



- Paxos and Raft use distributed state machines
- State machines are fully deterministic
- Committed operations are executed by all state machines in the cluster in the same order

Example operation: set variable *x* to value 5

Paxos

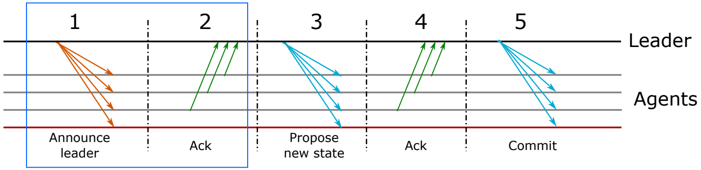
Old and well-known family of protocols for distributed consensus.

- first presented in 1989, paper published in 1998[6]
- has been formally proven to be safe

Complex and difficult to implement

- Basic Paxos protocol decides on a single output value
- Multi-Paxos extends the Basic protocol for practical use
- Many variants: cheap, fast, generalized, byzantine
- No reference implementation
- Many papers try to make it more approachable[8, 7, 3]

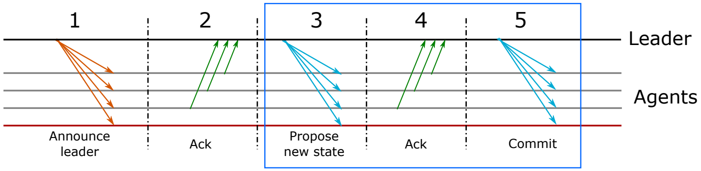
A Paxos round - 1



Phase 1 of a very simplified Paxos round:

- A node in the cluster self-appoints leader and chooses a new ballot ID
- Sends a ballot proposal to the other nodes (1)
- The other nodes return the highest ballot ID they know (2)
- If a majority responds with the proposed ID, the protocol can proceed
- The other nodes can include also their proposal for the value

A Paxos round - 2



Phase 2 of a very simplified Paxos round:

- The leader resolves any conflicting proposal for the value
- The leader proposes the new value (3)
- The leader receives the answers (4)
- If a majority accepts the new value, it is final for this round (5)
- Otherwise a new round must be started

Paxos - conclusions

- The original Paxos defines the protocol for one output value
- Multiple rounds can happen at the same time, even in single Paxos
- Paxos is complex to understand and implement
  - There are five different roles (only two are shown)
  - The leader can decide to work with a subset of the available nodes (a quorum)
  - Even in the same round there can be multiple proposals, each with an identifier
  - ... and we haven't even mentioned the state machine ...

Raft

Introduction

Modern implementation of a consensus protocol

- Published at Usenix 2014[9]
- Has been built to be easy to understand and implement

Random facts:

- The authors released a reference C++ implementation (LogCabin)
- Many implementations in other languages
- Lots of educational material, lectures and assignments are available[3]

[3]These slides are based on the original Raft paper[9] and the slide deck from Michael Freedman (Princeton, COS-418).

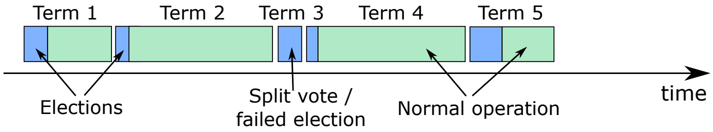
Introduction

- Raft maintains a distributed log containing state machine commands
- The *elected leader* handles all communication with the clients ($\rightarrow$ no communication until a leader is elected)
  and has the control on which entries are committed to the log
- The other nodes are passive replicators
- Snapshotting is used to keep the log size limited
- Periodic heartbeats are used to check if nodes are alive
- All nodes are known in advance

Node states

- At any given time, each node is either:
  - Leader: handles client requests, manages the log
  - Follower: passively replicates the log and the state machine
  - Candidate: transition state used during elections
- Normal operation, with N nodes (N must be odd):
  - 1 leader
  - N-1 followers
  - 0 candidates

Election terms



- Time is divided into terms, each identified by a monotonically increasing ID
- Each node records the term he believes to be the current one
- All messages are labeled with a term ID $\Rightarrow$ term IDs are used to identify obsolete information

Elections

How an election starts:

- Each node has a (bounded) random timer, the *election timeout*
- If a node does not receive any leader heartbeat before the timeout, it starts an election

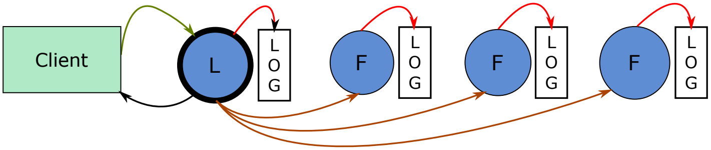
The node that starts an election:

- Increments the term ID, becomes *Candidate* and votes for itself
- Sends a request to vote to all other nodes until either:
    - Receives a majority of votes $\rightarrow$ becomes *Leader*
    - Receives a message from another leader $\rightarrow$ becomes *Follower*
    - No-one wins before the election timeout expires again $\rightarrow$ starts a new election

Election properties

- *Safety*: there is at most one winner/leader per term
  - Each voter votes only once per term
  - There cannot be two majorities in the same term
- *Liveness*: a leader will eventually be elected
  - The election is started after a random timeout
  - The randomness guarantees there will be different candidates at different times
  - Faults (hence new terms) happen in days/weeks/months, an election lasts milliseconds/seconds

Normal operation



1. Clients send commands to the current leader
2. The leader logs a new (uncommitted) entry
3. The leader sends the new entry to all nodes in the next heartbeat
4. Once a majority answers, the leader commits the new entry
5. The leader answers the client
6. The leader asks all nodes to commit the entry in the next heartbeat
7. The nodes commit the entry in their logs

What if ... the leader fails

Short answer:
easy, after the election timeout expires a new leader is elected

What happens to uncommitted entries?

- The client will never receive the commit ack (will retry later)
- Uncommitted entries in the followers will eventually be overwritten by the new leader

And when it restarts?

- It will restart as a follower
- It will set its internal term ID according to the first heartbeat message it receives
- The leader will send missing log entries

What if ... a follower fails

Short answer:
nothing happens

And when it restarts?

- Same as when the leader restarts
- It will restart as a follower
- It will set its internal term ID according to the first heartbeat message it receives
- The leader will send missing log entries

What if ... there is a 50/50 vote

Short answer:
no leader is elected, election timeout expires, new term, new election

This case is called *split majority*:

- An even number of voters
- Two equal subsets equal in size vote for different leaders
- The two candidates count the votes and see that there is no majority

Very low probability:

- two candidates at the same time (random election timeout)
- messages to initiate election arrive at the same time

What if ... there is a network partition

Short answer:
the majority rule ensures only half of the partition commits new entries

What happens:

- The biggest partition will elect a leader, incrementing the term ID
- The smaller one will be unable to do anything: all operations require a majority

And when the partition is healed?

- The minority partition will receive heartbeats with a bigger term ID
- The leader in the minority partition (if any), will immediately step down
- Uncommitted log entries will be overwritten

Learning tools and Raft summary

Raft interactive demos:

1. Raftscope: `http://bigfoot-m2.eurecom.fr/raftscope/`
2. Secret lives of data: `http://thesecretlivesofdata.com/raft/`

- Raft is a consensus protocol
- Consistently replicates a distributed log of state machine commands
- In each term a leader is elected
- Hard timeouts drive elections and heartbeats
- The leader is responsible for client communication and log replication

Some implementations of consensus protocols

- ZooKeeper (ZAB)
- Consul (Raft + Serf)
- etcd (Raft)
- OpenReplica/ConCoord (Paxos)

ZooKeeper

Motivation

When building a distributed system you have two options:

- Build your own coordination primitive each time
    - Buggy and error-prone approach
- Use an external coordination system
    - Adds external dependencies, more complex deployments
    - Does not reinvent the wheel
    - Use a well-known and tested system

Recent examples of coordination systems:

- Chubby from Google[2] (lock service)
- Centrifuge from Microsoft[1] (Lease service)

History

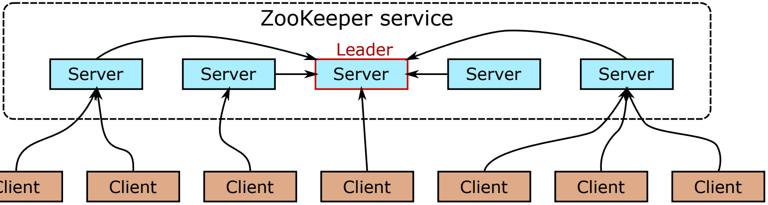
ZooKeeper was originally (2008) part of the Hadoop suite, since 2011 it is a stand-alone Apache project

Objectives[4]:

- Provide common services needed by distributed systems
  - Configuration
  - Group management
  - Naming
  - Presence protocols
  - Distributed synchronization
- Have a simple interface
- Have a highly available architecture

[4]See also the Tao of ZooKeeper:
`https://cwiki.apache.org/confluence/display/ZOOKEEPER/Tao`

Architecture



- ZooKeeper itself is distributed, for performance and fault tolerance
- Uses the ZAB consensus protocol
- Clients can talk to any server, but all update operations are handled by an elected leader
- Data is kept in-memory for performance
- Snapshots and transaction logs on persistent storage

Architecture - ZAB

The ZAB is the consensus protocol used by ZooKeeper

- It does not use state machine replication like Paxos or Raft
- Totally orders write requests using a majority of ZooKeeper processes
- Leader sequences the requests and invokes ZAB atomic broadcast
- Strictly ordered state updates are applied by non-leaders

ZooKeeper developers focused on strong ordering guarantees for all operations[5]

ZooKeeper and the CAP theorem

ZooKeeper is AP by default:

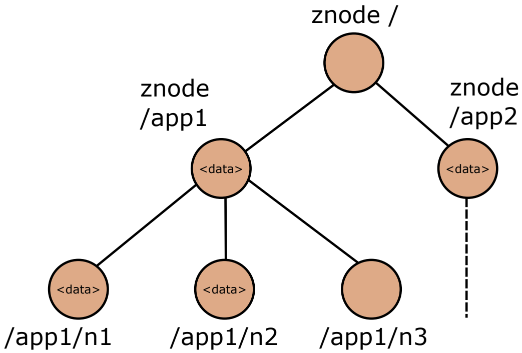
- reads return local (cached) data that may be stale

The client may use the sync command before a read:

- It makes ZooKeeper flush all caches
- Performance penalty
- Makes ZooKeeper CP

Note: writes are always synchronized through the leader.

Data model



- Hierarchical namespace (like a file system)
- Each data node is called *znode*
- Any znode can contain data and have children znodes

znodes

Znodes can be:

- *Regular*: created and destroyed by the client
- *Ephemeral*: created by the client, but ZooKeeper will delete it if the client disconnects
- *Sequential*: created by the client, but the name is generated by ZooKeeper using a counter
- *Ephemeral + Sequential*: combines the two above

Znodes have version counters that are updated each time their content changes

Watches

A client can set a *watch* on a znode. ZooKeeper will call back the client when:

- The data in the znode changes
- A children node is created or destroyed

Watches are very useful to implement locks and leader elections with good performance

Implementing consensus

Briefly:

- A number of processes need to agree on a value
- Each one proposes a value
- The decision must be unanimous

Each process proposes:

```
create(PATH, my_value, SEQUENTIAL)
```

Each process decides:

```
C = getChildren(PATH)
> Select znode z in C with smallest sequence suffix <
agreed_value = getData(PATH + z)
```

Implementing configuration management

Briefly:

- A number of processes need to access common configuration
- The config can change dynamically

Each process does:

```
CONFIG_PATH = /app/config
config = getData(CONFIG_PATH, watch=TRUE)
while (TRUE) {
    > wait for watch notification on CONFIG_PATH <
    config = getData(CONFIG_PATH, watch=TRUE)
}
```

Implementing group membership

Briefly:

- A number of processes provide the same service (load balancing)
- Leverage ephemeral znodes

Each process joins the group:

```
create(GROUP_PATH + proc_name,
       [address], EPHEMERAL)
```

Clients list the group members:

```
getChildren(GROUP_PATH + proc_name, watch=TRUE)
```

The watch is used to get notified about membership changes

Conclusions

- ZooKeeper is a high-performance coordination service for distributed applications
- Internally uses the ZAB consensus protocol
- Clients manipulate data in form of hierarchical znodes, similar to a file system
- Complex, higher level primitives can be built on top of the ZooKeeper API

ZooKeeper documentation:
`https://zookeeper.apache.org/doc/r3.4.9/`

Laboratory session - leader election

Context:

- Sensors are streaming data to your cluster
- Need a central node to do aggregations, the system must be highly available
- Implement the leader election algorithm on top of ZooKeeper

Objective:

1. Start three processes
2. One of them will become the active leader
3. Stop/crash the leader
4. One of the surviving processes automatically takes the lead
5. The crashed node restarts without disrupting the current leadership (bonus question)

References

References I

[1] ADYA, A., DUNAGAN, J., AND WOLMAN, A.
Centrifuge: Integrated lease management and partitioning for cloud services.
In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 1–1.

[2] BURROWS, M.
The chubby lock service for loosely-coupled distributed systems.
In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

References II

[3] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J.
Paxos made live: An engineering perspective.
In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[4] GILBERT, S., AND LYNCH, N.
Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.
SIGACT News 33, 2 (June 2002), 51–59.

[5] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M.
Zab: High-performance broadcast for primary-backup systems.
In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 245–256.

References III

[6] LAMPORT, L.
The part-time parliament.
ACM Trans. Comput. Syst. 16, 2 (May 1998), 133–169.

[7] LAMPORT, L.
Paxos made simple.
SIGACT News 32, 4 (Dec. 2001), 51–58.

[8] LAMPSON, B. W.
How to build a highly available system using consensus.
`http://research.microsoft.com/en-us/um/people/blampson/58-Consensus/WebPage.html`.

[9] ONGARO, D., AND OUSTERHOUT, J.
In search of an understandable consensus algorithm.
In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

References IV

[10] ROBINSON, H.
Consensus protocols: Two-phase commit.
http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit/.