

Dec 4, 2016

# Distributed systems cheat sheet

Since last year, I've been dealing with multiple distributed systems and suffering from a lack of structured understanding of how it all work under the hood. This post aims to break the subject down into small parts and aggregate multiple sources and links I found useful. It covers mostly distributed data storage systems but some concepts may be applied to other types of systems.

## Disclaimer

I'm not an expert in distributed systems. This blog post reflects my current understanding of the subject. If you find any mistakes, you're welcome to leave comments.

## What a distributed system is

A distributed system is a software working on multiple computers/VMs/containers (further in this article I'll call all of this a **node**) and communicating with each other via network interfaces. It helps solving scalability and reliability problems by using the following techniques:

- **Data replication** is copying of data to multiple nodes.
- **Data partitioning** is when data is divided into N parts and processed separately on several nodes.

Software Engineer with proven expertise in object-oriented analysis and design and exceptional record overseeing all facets of Software Development Life Cycle, from analysis and design to implementation and maintenance

Let's dive into this more deeper:

- Physical factors
- Time and Order
- CAP
- Consistency
- Replication

- Synchronicity
  - Synchronous
  - Asynchronous
  - Semi-synchronous
- Node relationship
  - Master/Slave (single copy)
  - Multi-master
- Consensus
  - Two-Phase commit
  - Three-Phase commit
  - Paxos
  - RAFT
- Partitioning
  - Horizontal partitioning
- Links

## ##Problems

There are two different worlds - theoretical model and practical experience. They both don't meet each other in most cases because these models describe a perfect environment, but there is an infinite number of problems that can cause a divergence.

## Physical factors

All nodes are connected to each other via the network and due to the many factors, this doesn't provide strong guarantees of data transferring and accessibility. Fallacies of distributed computing ([https://en.wikipedia.org/wiki/Fallacies\\_of\\_distributed\\_computing](https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing)) explains it as follows: Network is not reliable. Packages may be lost, reordered, or received multiple times, with different non-zero delays and limited throughput of the network. Moreover, the topology of the network is not constant - it can be changed at any time intentionally or accidentally due to network failures, also nodes can crash or be rebooted.

## Time and Order

When we have a sequence of events occurred in a distributed system, it's important to

know in what order they came. We can't rely on physical clocks because it's always not perfectly synchronized. That's why people came up with logical clocks.

Logical clocks are not about actual time, it's about an order of events, it's also called a partial ordering. In this model, all events are connected by happens-before relation (<https://en.wikipedia.org/wiki/Happened-before>). There are two implementations of this model:

- Lamport timestamps
- Vector Clocks

Here is a more formal description of this concept (<http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf>).

## CAP

All problems described in previous two sections can be combined in one model called **The CAP theorem**. It defines the relation between three properties:

- **Consistency**, guarantees that all nodes will see the same data representation;
- **Availability**, guarantees the fastest availability of the data ignoring its accuracy;
- **Partition tolerance**, the system survives network failures.

The theorem states that only two of these three can be chosen in the system. Knowing that the network with no failures and delays doesn't exist, we have only two types of systems to choose: CP and AP. Although in the real world, popular distributed systems are neither CA nor AP because they can provide different guarantees on demand or weaker consistency guaranties or even loose availability/consistency under certain conditions (<https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html>). This brings us to the idea that the CAP theorem in most cases is a high-level abstraction that can't be applied to the systems you write or use but can be a starting point for understanding more complex cases.

## Consistency

It's important to know that the **Consistency** in the CAP theorem means strong

consistency, specifically linearizability. But there are many different types of consistency.

- **Strong consistency** all nodes see the values in the same order
  - *Strict consistency*: (impossible to implement in a distributed system) It orders all writes and reads relying on absolute global time.
  - *Linearizability*: the order of reads on all nodes matches the order of writes. A read can obtain the most recent applied value or a value that goes after it.
  - *Sequential consistency*: the order of reads doesn't match the order of writes but is the same for all read operations.
  - *Causal consistency*: the order of reads of causally-related writes matches the writes order.
- **Weak consistency** the order of reads can differ
  - *Client-centric consistency*: the client works with the distributed system through a cache or uses a session. Unless client writes then reads either through the same cache or session bound to the same node, all reads will be consistent, while other nodes can see an inconsistent state.
  - *Eventual consistency*: "It defines that if no update takes a very long time, all replicas eventually become consistent"

Here are [useful slides \(http://www.cs.cmu.edu/~srini/15-446/S09/lectures/10-consistency.pdf\)](http://www.cs.cmu.edu/~srini/15-446/S09/lectures/10-consistency.pdf) about consistency models.

## Replication

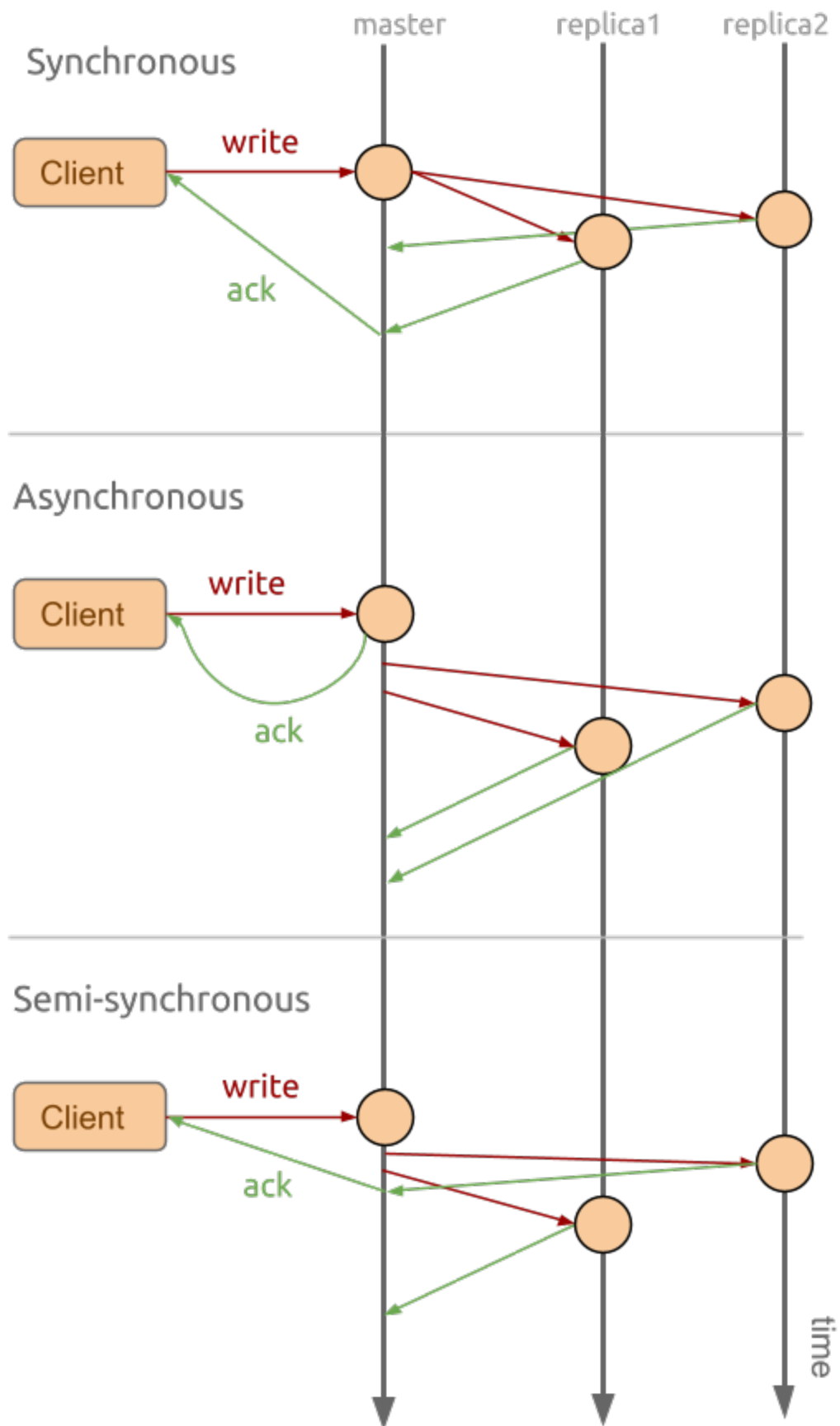
Now we can take a look at more practical aspects of distributed systems. Let's assume we have data stored in a system working on one node, now we want to have:

1. fault-tolerant: if one node is down, another one can replace it
2. higher accessibility: if we have too many reads and clients are experiencing difficulties under high load, then we copy data to another node to spread the load

Both of these goals can be achieved by multiple approaches depending on more detailed requirements.

## Synchronicity

The actual copying of the data during the replication can be conducted in several manners.



## Synchronous

When a client writes some data to the master node, the node accepts data and saves it to its own storage plus sends this data to all replicas. Once all replicas responded back with OK, the master node returns a response to the client. If data didn't do through to any of replicas, the client will receive an error code - this guarantees data consistency.

Since the master should wait the response from all of the nodes, a single write operation cannot be faster than the time of the slowest combination of *data write to a node + network latency for this node*

When one of the nodes becomes inaccessible, the master stops accepting writes - all nodes are only available for reads.

## Asynchronous

In this replication approach, a client receives response as soon as master thinks that the data has been handled by the master, the data copying to replicas is starting asynchronously. The client won't be notified about copying failure. Asynchronous replication offers fewer guarantees but more faster writes from the client point of view.

If the master goes down and some data hasn't been properly delivered to the slaves, this data will be lost even though the client is aware of data write.

## Semi-synchronous

There is a way to combine synchronous and asynchronous approaches called semi-synchronous replication. This approach works the same way as synchronous but returns response once at least one replica acknowledged.

In case of single replica, this approach is equal to synchronous.

## Node relationship

### Master/Slave (single copy)

The most standard replication model is master/slave replication. In this technique, there should be one master node and single or multiple slaves. Master accepts writes and reads but slaves accept only reads. When master goes down, replicas stay available for

reads.

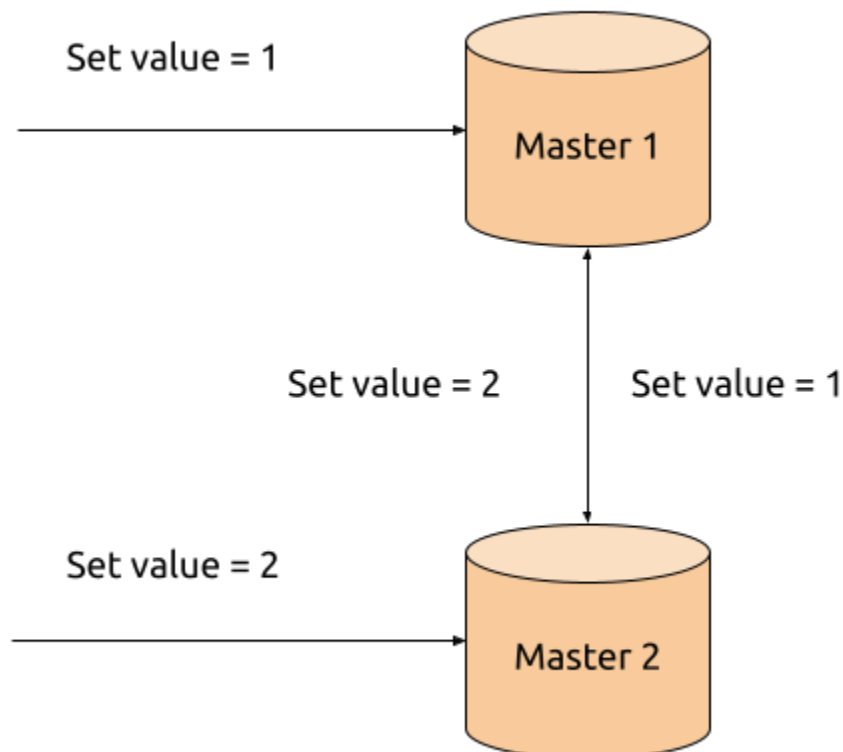
Usually master sends operation log (binary log) or a log with global transaction identifiers, this allows all replicas understand where they are at in that log and catch master up after a failure or initially.

Most real-world systems promote a slave to a master when the master becomes inaccessible.

### Multi-master

Multi-master is where you can write on any node and data gets replicated to all others. This addresses the problem of the many writes in HA systems.

Usually, this approach causes different issues due to the fact that clients can work with the same data but from different masters. The following techniques can solve this problem:





1. **Pessimistic locking** - master locks a potentially affected segment of data so others master refuse modification of the segment.
2. **Optimistic locking** - based on data versioning. If a master A tries to write a value with a version that doesn't equal to the one that a master B has, this write will be refused.
3. **Conflict resolution**
  - Priority - from two conflicting writes we apply a one by some rule (e.g. the write with the latest timestamp)
  - Data merging - convergent or commutative replicated data type (<http://hal.upmc.fr/file/index/docid/555588/filename/techreport.pdf>)

**Martin Kleppmann - Conflict Resolution for Eventual Consis...**



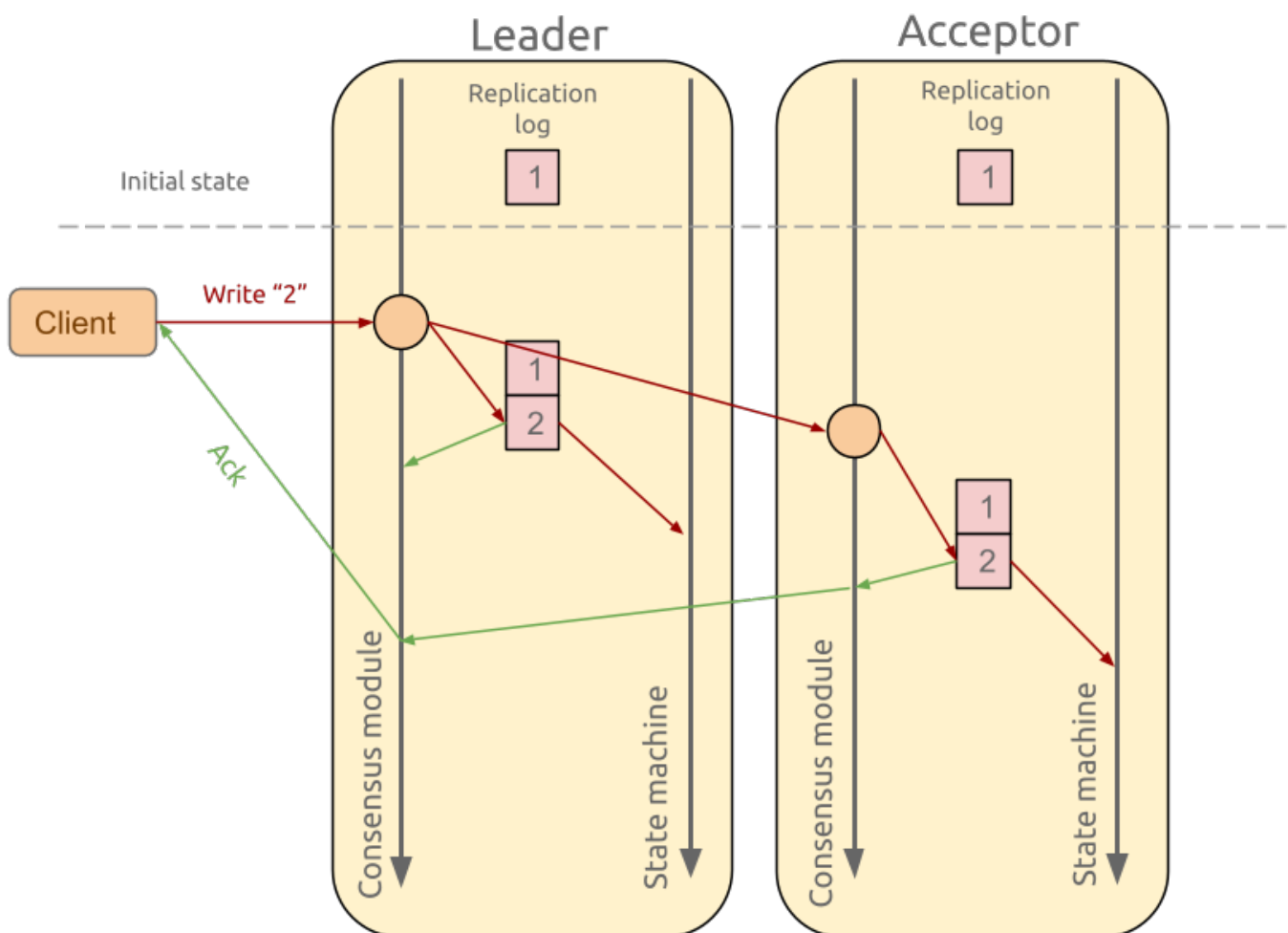
1. **Conflict avoidance** - data is divided into logical parts and assigned to specific master, so clients are aware of what type of data should go to a specific master.  
**This approach is very similar to partitioning.**

## Consensus

The consensus problem is a problem of having nodes in distributed systems agreed on some value.

Consensus typically arises in the context of replicated state machines, a general approach to building fault-tolerant systems. Each server has a state machine and a log. The state machine is the component that we want to make fault-tolerant, such as a hash table. It will appear to clients that they are interacting with a single, reliable state machine, even if a minority of the servers in the cluster fail. Each state machine takes as input commands from its log. In our hash table example, the log would include commands like set x to 3. A consensus algorithm is used to agree on the commands in the servers' logs. The consensus algorithm must ensure that if any state machine applies set x to 3 as the  $n^{\text{th}}$  command, no other state machine will ever apply a different  $n^{\text{th}}$  command. As a result, each state machine processes the same series of commands and thus produces the same series of results and arrives at the same series of states.

This may be illustrated as follows:



## Two-Phase commit

The Two-Phase commit algorithm introduces two roles:

- Leader (coordinator, master) - a node that a client writes to.
- Acceptor (cohort, replica) - a node that all writes should be replicated to.

This algorithm is very simple and relies on the assumption that network is reliable.

There are one leader and one or multiple acceptors in the system. When a leader receives the data to write, it proposes the data to all acceptor and waits for a vote from all of them. An acceptor can accept or reject the value base on acceptor's state, then:

- If all acceptors accepted the value, the leader sends a commit request.
- If one or more of the acceptors failed, the leader sends a rollback request.

After that, the leader receives acknowledgment for all commit/rollback command and returns to the client. More detailed description of the steps can be found [here](http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit/) (<http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit/>).

**This approach has a critical disadvantage** - it's not partition tolerant and it's blocking. If the leader fails in the middle of the agreement, the acceptors will be infinitely blocked on waiting for commit/rollback request. If some of the replicas fail permanently, then master won't be able to accept any writes.

## Three-Phase commit

The Three-Phase commit algorithm is an improved version of Two-Phase commit, that aims to eliminate its blocking nature. The algorithm introduces additional revertible phase of pre-commit where acceptor can store the data and switch to the state of waiting for a commit request. If an acceptor doesn't receive a commit request within specified timeout, the data will be discarded. Even though it solves the problem of blocking, the system can get in an inconsistent state.

## Paxos

Paxos is the next generation of consensus algorithms. It solves the weaknesses of the

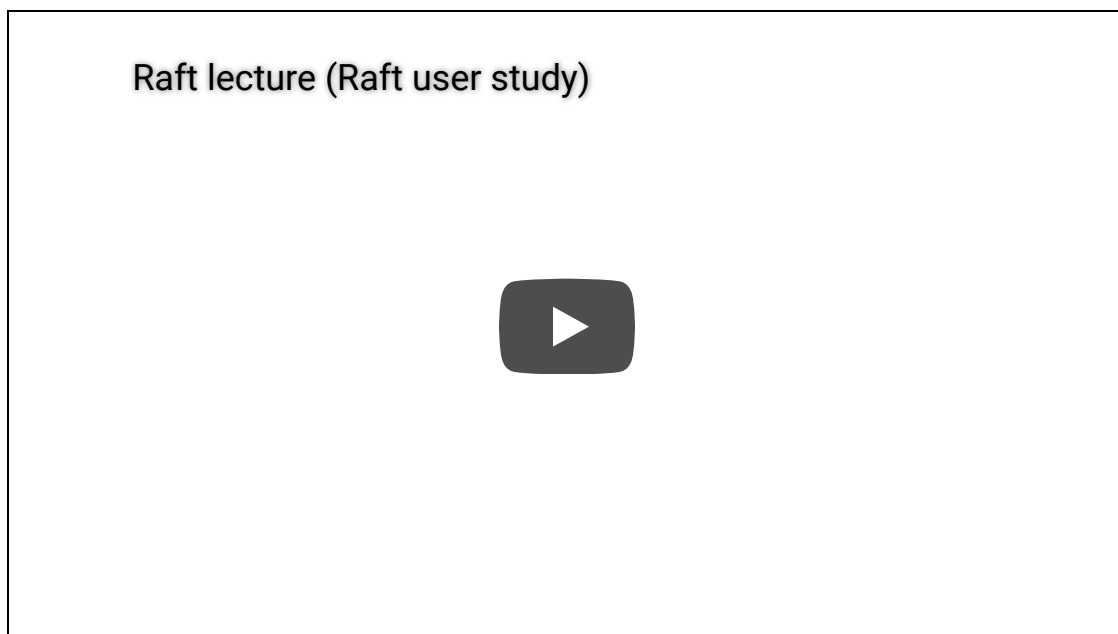
Two-Phase commit and Three-Phase commit algorithms. The main differences are that Paxos requires the majority of nodes to acknowledge in order to send a commit request, in addition to that, the leader assigns a unique sequence number to each write so that an acceptor can distinguish which value comes first.

You may look at a more detailed description [here](http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf) (<http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf>) or [here](http://the-paper-trail.org/blog/consensus-protocols-paxos/) (<http://the-paper-trail.org/blog/consensus-protocols-paxos/>).

There are also multiple modifications of the original paxos algorithm like Multi-Paxos, Cheap Paxos, or Fast Paxos.

## RAFT

Nevertheless, Paxos is not the only partition tolerant algorithm. Furthermore, Paxos is blamed for its complexity. The Raft algorithm is the youngest from all described above. It's relatively simple and give the same guarantees as Paxos does.



The best entry point to the details of this algorithms is [this site](https://raft.github.io/) (<https://raft.github.io/>).

## Partitioning

Partitioning is a technique of dividing the monolith data into several parts and storing them on different nodes. This technique helps to deal with large data that doesn't fit one node or deal with a big number of writes/reads when one node of the system can't handle it. Sometimes it can be used for security purposes (if data should be isolated in a certain way) or when geographical location of data parts is important.

There are three strategies of partitioning:

- **Horizontal partitioning** (sharding) - having a set of objects, each object depending on some rule is stored on the specific node.
- **Vertical partitioning** - having the same set of objects, each node stores only specific attributes of each object (e.g. user name is held by node A while user DOB is held by node B).
- **Functional partitioning** - this strategy divides the data by type (e.g. user info and user orders are located on different nodes).

All these strategies solve a particular problem but bring other problems. The main problems are:

- Probable inconsistency. Data on different nodes are not linked so that we can lose some data.
- More complex and time-consuming data querying. If the data we need is located on different nodes, we need two steps to obtain the result: query all nodes and then merge the results.

## Horizontal partitioning

Horizontal partitioning is the most popular strategy due to the fact that in most cases distributed systems deal with a big number of homogenous objects. Also, this approach gives more scaling flexibility.

The most interesting part of this strategy is the rule that will be used for data division. This rule is called "sharding key". It should be chosen beforehand (because it's relatively difficult in most cases to change this key after the system is in use) and should:

- Divide data into approximately the same parts so that there won't be nodes with

too much data and with too little data.

- Divide data into parts with approximately the same load. It means that hot data shouldn't be concentrated on a certain node while unused data on another node.
- Divide by geography. Data related to specific location should preferably be located in the corresponding geographic data-center.
- Data of a node A shouldn't avoid relations to data of other nodes.

A sharding key can meet one or more of the properties below depending on the problem. There are several common solutions to choose a sharding key:

- **Range partitioning**, the key will be the interval of the value of some sequential attribute (e.g. date of the transaction, user DOB, etc).
- **List partitioning**, the attribute value itself (e.g. user country).
- **Hash partitioning**, the key is calculated by applying a hash function to any of model's attributes.
- **Dynamic partitioning**, keys are associated with values by a certain rule and these relations are stored in an external system.

## Links

1. [Awesome Distributed Systems \(https://github.com/theanalyst/awesome-distributed-systems\)](https://github.com/theanalyst/awesome-distributed-systems)
2. [Outline for Aphyr's class "Distributed Systems Fundamentals" \(https://github.com/aphyr/distsys-class\)](https://github.com/aphyr/distsys-class)
3. [Distributed systems for fun and profit \(http://book.mixu.net/distsys/single-page.html\)](http://book.mixu.net/distsys/single-page.html)



(mailto:?subject=Distributed%20systems%20cheat%20sheet&body=http%3A%2F%2Fdimafeng.com%2F2016%2F12%2F04%2Fdistributed-systems%2F)



(https://twitter.com/share?url=http%3A%2F%2Fdimafeng.com%2F2016%2F12%2F04%2Fdistributed-systems%2F&text=Distributed%20systems%20cheat%20sheet)

**f** (<https://facebook.com/sharer/sharer.php?u=http%3A%2F%2Fdimafeng.com%2F2016%2F12%2F04%2Fdistributed-systems%2F>).

**G** (<https://plus.google.com/share?url=http%3A%2F%2Fdimafeng.com%2F2016%2F12%2F04%2Fdistributed-systems%2F>).

**in** (<https://www.linkedin.com/shareArticle?mini=true&url=http%3A%2F%2Fdimafeng.com%2F2016%2F12%2F04%2Fdistributed-systems%2F>).

#### ALSO ON DIMAFENG'S BLOG

##### **Dynamic bean definition for ...**

7 years ago • 1 comment

This blog is about  
programming tricks and tips.

##### **Netty Hello World**

6 years ago • 2 comments

This blog is about  
programming tricks and tips.

##### **Linux command line cheat sheet**

6 years ago • 4 comments

This blog is about  
programming tricks and tips.

##### **Mong within**

7 years

This bl  
progra

2 Comments

dimafeng's blog

 Disqus' Privacy Policy

 Login ▾

 Favorite 1

 Tweet

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 



Name




**anevseev** • 5 years ago

Very nice review, thank you.

I think there are two typos in the `Two-Phase commit.` part: `s/date/data`.

^ | ▾ • Reply • Share ›




**dimafeng** **Mod**  anevseev • 5 years ago

Thank you! Fixed :)

^ | ▾ • Reply • Share ›

 Subscribe

 Add Disqus to your siteAdd DisqusAdd

 Do Not Sell My Data