Fault Tolerance in Large-Scale Distributed System

Advance Data Management's Project

A.Y. 2022/2023, UNIGE

Student:

Edoardo Pastorino, 5169595

January 27, 2023

Contents

Contents 2

1 Introduction 3

2 What is Fault Tolerance? 4

2.1 Main Fault Tolerance's issues . . . 5

2.1.1 Failure Detection . . . 8

2.1.2 System Recovery . . . 9

3 Fault Tolerance's Approaches 11

3.1 What is Paxos? . . . 11

3.1.1 Selecting phase . . . 12

3.1.2 Learning phase . . . 14

3.1.3 Fault Tolerant Implementations and Issues . . . 15

3.2 What is Chord? . . . 17

3.2.1 Chord's Fault-tolerance . . . 19

3.3 Final Comparison between Paxos and Chord . . . 21

4 Fault Tolerance Management in NoSQL Systems 23

4.1 Neo4j . . . 23

4.1.1 System's Fault Tolerance Approach . . . 24

4.2 MongoDB . . . 27

4.2.1 System's Fault Tolerance Approach . . . 27

4.3 Cassandra . . . 31

4.3.1 System's Fault Tolerance Approach . . . 31

5 Conclusion 34

References 36

1 Introduction

This paper is related to my final project of the Advanced Data Management course of the computer science's master degree (track Data-centric Computing) at the university of Genoa (UNIGE), of the academic year 2022/2023.
The purpose of this document is to provide to the readers an overview of the most important fault tolerance aspects and approaches used in large scale distributed systems, enriched by some analyses about this topic also in some important NoSQL systems.
The section 2 is that relating to an introduction of the fault tolerance problem, with more details about the main issues that we have to face for providing robustness against failures.
Inside the section 3 I have tried to present two of the main approaches used in different distributed architectures for providing some fault tolerance features such as node failure robustness and leader election. This two methods are explained in the subsection 3.1, for what concern Paxos consensus protocol, and in subsection 3.2 regarding the Chord algorithm.
In the section 4, instead, I have decided to show what are the approaches, for providing fault tolerance, used in three of the most popular NoSQL distributed systems, Neo4j (subsection 4.1), MongoDB (subsection 4.2) and Cassandra (subsection 4.3).
In the section 5 I have concluded my presentation of this important topic, summarizing the used approaches in all the analyzed systems.
Some information reported in this project's report are been retrieved inside slides owned by my university, some others inside public online articles, papers and books on the subject, all found in the references of this document.

2 What is Fault Tolerance?

Fault Tolerance, in its most general meaning, refers to the capability of a certain system (computer system, cloud cluster, network, . . .) to deal with system failures, to continue operating even if one or more of its components fail ([Imp]). Therefore, a fault tolerant system is a system that is able to continue to perform despite some component's crash.
A fault is the failure of a component of the system and can be associated, for instance, to a node's crash, or to the happening of an error during execution of some operation. Faults can be very different depending on the system we are considering.
In a centralized system, a system in which the correct execution of operation is deterministic, the faults follow an atomic behavior: the system can works or not, so we talk about total failure (since we have a single point of failure). For this reason, usually the chances to have a single machine fail are low. This behavior is a design choice for single machine because we prefer a computer to crash than having a wrong result, since incorrect results are difficult to manage ([Kle17]).
In a distributed system, faults can occur with more frequency and, when they happen, some parts of the system could be broken, but some others could still working, so there is a partial failure. The complexity is that this kind of failure are nondeterministic, you don't know exactly when an operation that involves some nodes will work correctly or will fail. The atomic behavior is acceptable only for small tasks, while for long distributed tasks can't be accepted, since errors can occur too often. The nondeterminism and the partial failures are the difficult aspects that you have to take into account when you work with distributed systems.
There are two general properties for a large scale distributed system:

- **Safety property**, means that "nothing bad can happens". The violation occurs when we have an execution that leads to a bad state. The violation cannot be undone, the damage is already happened.

- **Liveness property**, stands for the fact that "something good eventually happens". In this case the violation happens when we entering a loop in which we can't reach good states. But in general, if an issue occurs, there is always the hope that it could be fixed in the future.

Then, we can have four main fault tolerance configurations based on the fact if these two properties are preserved or not:

- **Masking**, in which both the safety and the liveness are preserved.
- **Non-Masking**, where the safety property can be temporarily violated. This is not true for the liveness that is preserved, so also the safety is eventually restored.
- **Fail-safe tolerance**, the safety property is preserved, but liveness, this time, can be violated.
- **Graceful degradation**, it's a scenario in which application continues in a degraded mode and so everything depends on the level of degradation that we can, or we want to accept.

2.1 Main Fault Tolerance's issues

Since usually a large-scale distributed system is a shared-nothing architecture that uses a network for making communicate the machines, the most general and common failure is, in fact , the so called **Network fault**, that occurs when a part of the considered network is cut off from the rest. Since most of internal network in data centers are again shared nothing architectures, that follow an asynchronous packet network in which a node can send a message to another one, without the guarantees of when it will arrive, or whether it will arrive at all, a lot of things could go wrong during the message passing. Some issues could be for example that your request (but also the answer) could be lost (omission failure), or it could be delivered later (again as the answer), or the remote node could be crashed or stopped. Due to all these problems, if you send a request to another node without receive the associated response, you are not able to understand why, as the image below demonstrates: you can't know if the request was lost (a), or the response was lost (c), or the remote node was down (b).

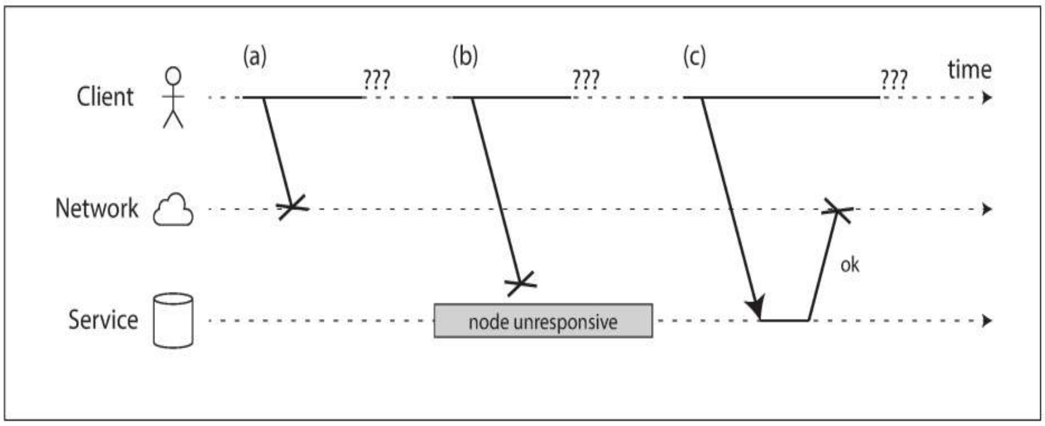

Figure 1: Problems in message passing.

The common method to handle this problem is a timeout-oriented approach: after a certain predefined period of time a client gives up waiting for the answer and assumes that it won't arrive, even if the client doesn't know exactly if the remote node has received the request or not.
The important aspect is that, in a distributed system, whenever any communication process happens through a network can fail and we must be able to handle the failures, testing our system to avoid that arbitrary unexpected bad things will happen. Deal with failures doesn't mean tolerate them, sometimes we can only show an error message without solving it.
Another different kind of fault that can occurs in a distributed system is the so called **Byzantine failure**. Previously we were based on the assumption that nodes in the network could be unreliable and so some message could be lost, but they were honest, so they told the truth if everything is correct or not. If now this assumption is no more valid, a distributed system can encounters this new particular fault. For instance a node can say that it have received a certain message when this is not true in practice. So a system is defined as byzantine fault-tolerant if it continues to operate

in the right way even if some nodes don't follow a specific protocol or are malicious. Byzantine fault tolerant system are however difficult to build and sometimes are so complicated that are too expensive in term of implementation. In addition, in situation in which there is a server (master-slave architecture for example), are quite useless, since the server has the authority for deciding if a node is reliable or untrustworthy. This is not true in a P2P scenario in which byzantine fault tolerance may be useful.
In a distributed system there are also others different types of specific failures that can occur like:

- **Crash failure**, occurs when a process halts forever and it is irreversible.
- **Transient failure**, from the hardware point of view it occurs when there is a perturbation of the global state due to weak batteries, lightnings or radio-frequency interference .... From the software point of view they are characterized by the so called heisenbugs ([Wika]), a software bug that seems to disappear or alter its behavior when one attempts to study it. In fact, Heisenbugs occur because common attempts to debug a program usually have the side-effect of altering the behavior of the program in subtle ways .
- **Software failure**, occurs when there are coding error, inaccurate modeling, memory leaks, so it can be caused by human errors. These failures can cause others kind of problems, like crash or omission issues.
- **Temporal failure**, occurs when it is impossible to respect a deadline due to poor algorithms and strategies or since the synchronization among processors clocks could be lost.

Therefore, the main fault tolerance's process' issues are how to detect that the system encounters a failure (failure detection) and how to make the system working again (system recovery).

2.1.1 Failure Detection

Many distributed systems need to have mechanisms for automatically detecting faults, like discovering when a node is dead and so stop to send requests to it, or, if the node is the master, in a typical single-leader replication scenario, re-elect a new node as leader. Sometimes, however, it is necessary to receive some feedback that explain explicitly that something is not working, but, since you have to assume that you will get no response at all when something has gone wrong, retry a few times, waiting for a timeout to elapse and after consider the node dead, can still a good solution. The classical network fault and the omission failure for non FIFO channels use this timeout-based method, like also the crash failure, in a synchronous scenario. Exist also other detection mechanisms for crash failure in an asynchronous scenario based on the fail-stop failure: an abstraction that simulates the crash failure stopping a component. The omission failure for a FIFO channel can be also detected using a sequence numbers for the messages: if a number in the received sequence is missing we can say that the associated message is lost. In the end transient failures could be found with the usage of global snapshots of the distributed system, confronting the current one with the previous to detect some violations.
Since the timeout-based approach is the most useful in lots of the previous situation, now the main question is: how long should the timeout be?

- A too **long timeout** means a long waiting time for the user and limits the availability of the system.
- A too **short timeout** means an high risk on declare prematurely dead a node that could suffer only a delay. This fact can be very problematic because if a node is declared erroneously dead while it is performing an action, another nodes will perform the same action in substitution, hence this action may end up being performed twice. Another problem could be the fact that declaring in too early dead nodes can cause a cascading failures in transferring the load from dead nodes to the others.

In the end we can say that there is no a perfect correct value for timeouts, they have to be determined empirically.

2.1.2 System Recovery

Recovery from an error is essential to fault tolerance. The two main recovery approaches after the occurrence of a failure are ([Geea]):

- **Backward Recovery**. In this recovery approach we want to move the system back to a previous correct state from the incorrect current one. We can perform this action periodically recording the system's state and restoring it when something goes wrong. There are in turn other two ways for performing backward recovery:
  - *Checkpointing,* is a method based on checkpoint, that are distributed global system snapshots taken periodically and stored on a stable storage. With snapshots we can return to the most current one, also known as a recovery line. In other words, as showed in the picture below, a recovery line represents the most recent stable cluster of checkpoints.
  - *Message logging,* also this other approach is based on checkpoints, but instead of restoring the global state saved on a stable storage, we can retransmit the communication since the last checkpoint. A message is called as logged if its data and index of stable interval are both recorded on stable storage. Now if transmission of messages is replayed, we can still reach a globally consistent state, recovering logs of messages and continue the execution.
- **Forward Recovery**. In this second method, instead of returning the system to a previous, check pointed, state, when we face an issue, an effort is made to place the system in a correct new state from which it can continues to operate. So the computation does not care about getting the history right, but moves on, as long as eventually the safety property will be restored.

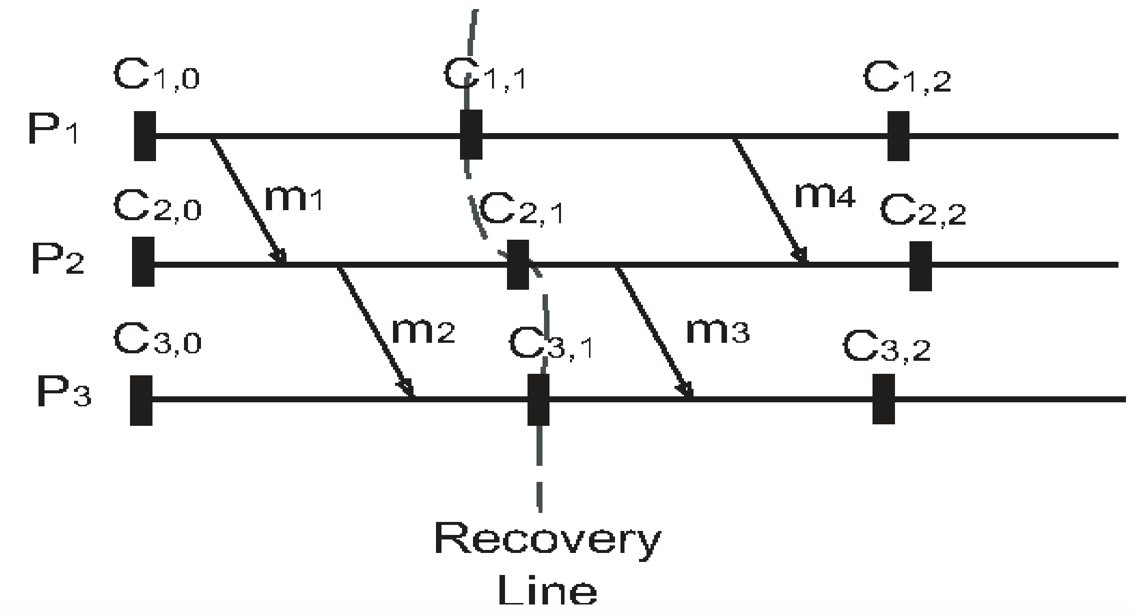

Figure 2: Checkpointing in distributed system

In a large scale distributed system, when we explain recovery, we have to talk about the (distributed) consensus protocols, since recovery usually is based on these protocols. A consensus protocol is a method, an algorithm for reaching agreement about which process has failed, for performing leader election, for implementing distributed mutual exclusion, ecc... . In the following section of this paper we want to focus on consensus protocol, but also general algorithm, related to fault tolerance.

3 Fault Tolerance's Approaches

In this chapter we dig more in details about some implementations of algorithms for fault-tolerance in a distributed system. Specifically we analyze and compare two famous and largely used protocols that can be also related to fault tolerance: the Paxos algorithm and the Chord algorithm.

3.1 What is Paxos?

Paxos ([Lam01]) is a family of protocols for solving consensus in a network of unreliable or fallible processors. Consensus can be defined as agreement, protocols are rules. Simply, consensus protocols could be viewed as "agreement rules". Consensus protocols are the basis for the state machine replication approach to distributed computing, where state machine replication stands for a technique to convert an algorithm into a fault-tolerant distributed implementation.
The main goal of the Paxos algorithm for implementing a fault-tolerant system is try to ensure that a single value, among all the values proposed by a collection of processes, is chosen. If a value has been chosen all the other processes should be able to learn the specific chosen value, on the contrary, it is also possible that no value is chosen if no value is proposed.

In this algorithm there are three main safety rules that have to be respected:

- Only a single value can be chosen at the end of the procedure.
- Only a value, among the ones proposed by the collection of processes, can be chosen.
- The processes learn the chosen value only when it is actually chosen.

The algorithm also ensures some eventually conditions: that a proposed value is eventually chosen and then that a process eventually learns this chosen value. In

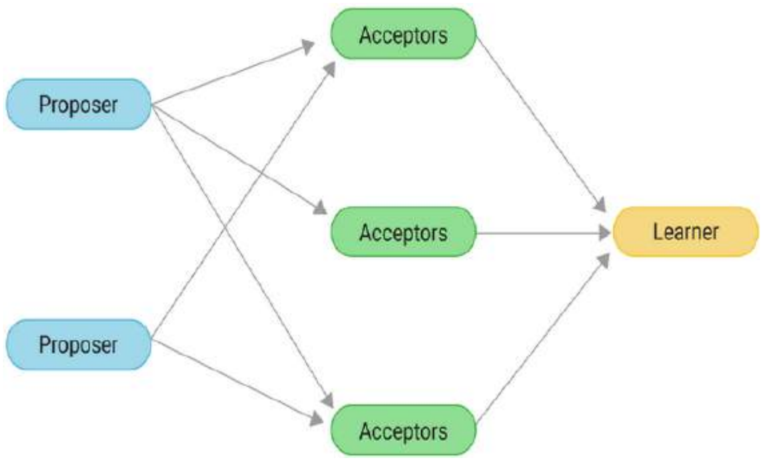

Figure 3: Paxos agents

this scenario we can associate processes to agents, divided them in three main categories: proposers, acceptors and learners. Each process (agent) can belong to more than one of these categories. We also assume that all the agents could fail after the selection of a value, therefore a solution is impossible to achieve without recording some information. Also the messages exchanged among the agents could be lost, duplicated or delivered with a certain delay, but never corrupted, since we consider a non-Byzantine faults scenario, in which, as we said in the previous chapter, all the agents are reliable.

3.1.1 Selecting phase

The **selecting value** procedure of the algorithm is composed by different requirements:

- **P1. An acceptor must accept the first proposal that it receives**, since a single proposer can propose a single value. But this requirement has the problem that, even with two only proposed values, if each is accepted by half of the acceptors agents, the crash of a single acceptor can create a situation in which it is impossible to decide what is the selected value.

  - *P1-a. An acceptor can answer to a proposal numbered n iif it has not responded yet to a prepare request with a number grater than n.*

- **P2. If a proposal with a certain value v is chosen, then every others proposals, associated with an higher number, that are chosen have the same value v.** Each proposal has an associated value v and a number n.

  - *P2-a. If a proposal with a value v is chosen, then every others proposals, associated with an higher number, that are accepted by any acceptor, have value v.*
  - *P2-b. If a proposal with a value v is chosen, then every others proposals, associated with an higher number, that are proposed by any proposer, have value v.* Now we are in a situation in which any proposed proposal has the value v, as, in turn, any accepted proposal, as well as, in the end, any chosen proposal. We are able to say that P2-b implies P2-a that in turn implies P2.
  - *P2-c. For any v and n, if a proposal with value v and associated number n is proposed, then there is a set S of acceptors such that: 1) or no acceptor in S has accepted a proposal with a number less than n; 2) or v is the value of the proposal with the high number among all the others proposals with a number less than n, that are accepted by the acceptors agents in S.* For satisfying P2-b we have to maintain the invariance of P2-c.

Paxos' algorithm can allow to execute two different kind of requests from the proposer:

- **Prepare request**, is the request done by a proposer to acceptors, asking them to answer with: a promise that they never again accept a proposal with an associated number less than n together with the proposal with the highest number less than n accepted 'till present.

- **Accept request**, is another request performed, after the prepare request, by the proposer in which it asks to a set of acceptors to accept its proposal. This proposal contains the value v that is the value of the highest-numbered proposal between all the proposals received as answer in the previous prepare request.

And it is divided in two phases:

- **Phase 1:** a proposer issues a proposal targeted with a number n and sends a prepare request to a set composed by the majority of acceptors. If this set of acceptors has not received yet another request with an higher number n, it accepts this request and answers to the proposer with the promise that further proposals with a number smaller than n will be never accepted. It also sends to the proposer in question the higher-numbered proposal 'till now.

- **Phase 2:** now the proposer can ask, with an accept request, to each of the previous acceptors to accept its proposal associated to the number n for the value v, which is the higher-numbered proposal's value among all the responses received in the previous phase 1. Of course an acceptor, following what we have discussed until now, accepts a proposal only if it has not responded to an accept request for a proposal with a number greater than n.

3.1.2 Learning phase

A learner agent is a process that want to find that a proposal has been accepted. The **learning value** section of the algorithm can be, instead, performed in three various approaches:

- **Method 1**, this first configuration is the simplest one in term of implementation. Each acceptor sends the answer with the chosen value to all the learners agents. In this way for a learner is easy and quick obtaining the information of the selected value, but requires a number of responses equal to the product of acceptors x learners.

- **Method 2**, in the second method, instead, each acceptor sends the answer with the chosen value to a single distinguished learner. Since we are in a non-Byzantine assumption, the learner can exchange messages among them for informing each other of the selected value. This time, the number of responses are the sum of the acceptors + learners, but there is an additional step for the learners to achieve the chosen value. There is also a possible issue in term of robustness, since the distinguished learner is a single point of failure for the communicating acceptor.

- **Method 3**, The third approach is the one used in practice because is more robust in terms of faults and doesn't have a too large number of responses. In fact, each acceptors sends the answer with the chosen value to a set of distinguished learners (no more single point of failure and no more responses equal to the learners x acceptors).

3.1.3 Fault Tolerant Implementations and Issues

The simplest consensus protocol could be the one that will use a single acceptor. Multiple proposers will concurrently propose various values to a single acceptor. The acceptor chooses one of these proposed values. Unfortunately, this does not handle the case where the acceptor crashes. If it crashes, after choosing a value, we will not know what value has been chosen and will have to wait for the acceptor to restart. In fact, to achieve fault tolerance, we will use a collection of acceptors. Each proposal will be sent to at least a majority of these acceptors. If a majority of them chooses a particular proposed value, then that value will be considered chosen. Even if some acceptors crash, others are still working, so we will still know the final result. If an acceptor crashes after it has accepted a proposal, other acceptors know already that a certain value was chosen. The system requires 2m+1 servers to tolerate the failure of m servers. In this way the protocol is able to deal with some failure-cases:

- Acceptor fails during phase 1: if the proposer still receive responses from the majority of the acceptors agents the protocol doesn't suffer from any problem and can go on.

- Acceptor fails during phase 2: this could not be a problem if, again, the majority of acceptors are still working and so the proposer and learners know the chosen value.
- Proposer fails after prepare request: an acceptor would have sent responses back doesn't receive any accept request, so there would be no consensus. Some other node will eventually runs as a proposer, proposing a value v with a number n. If the number is highest, then the algorithm runs, otherwise the request is rejected and the proposer have to make a proposal with an higher number.
- Proposer fails during accept request: some other node will eventually runs as a proposer and proposes a value v with number n in a prepare request. The acceptors answer informing the new proposer that a proposal was already accepted, and so the new proposer propagates now the old value completing the process.

The Paxos algorithm can also be used in practice, in a fault tolerant system, to elect a leader process that is going to play the distinguished proposer and learner in the overall procedure when the previous leader fails, so works well in a single leader scenario. We can imagine as value v, of a proposer that wants to became a leader, the sentence "I am the leader ([Wikb]).
In this algorithm the messages targeted as request and as response are sent as network messages, while all the information needed to be remembered by acceptors are stored in a stable storage, as well as the information used by the proposers.
There are unfortunately some problems related to the Paxos algorithm. For instance it is possible that two proposers continue to propose proposals with increasing numbers without any of these proposals being chosen, because a proposer could complete phase 1 with a number n1, while another proposer also could complete the same phase for another number n2 greater than n1 and again the first proposer, who

saw is proposal rejected, could do the same for another number n3, this time greater than n2 and so on and so for. Thanks to a single distinguished proposer as the only one that can issue proposals the so called progress is guaranteed. Another problem could be the fact that two proposer chose the same number associated to their own proposal. This is, fortunately, impossible to happen since each proposer chooses value from disjoint set of numbers.

3.2 What is Chord?

The Chord protocol ([IS03]) is a lookup protocol that try to support, as main functionality, the efficient location of data items inside the nodes in a peer to peer distributed system. This method is based on consistent hashing for assigning the data items, using their keys as input of an hash function, to the nodes.
In the consistent hashing scenario the nodes and the data are stored in the same space address, that can be viewed as a clockwise order scanned ring modulo 2^m, where m is a predefined number of bits, big enough to make the probability of two keys to end up in the same node very small. So the possible hash values can fall in the interval $[2^m - 1]$.

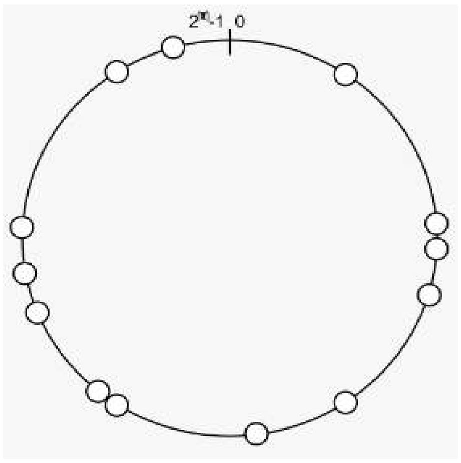

Figure 4: Chord Ring

For assigning key with consistent hashing in this ring space address we follow

this rule: an hypothetical key k is is assigned on the first successor node of k, this means that this node is the first one that has the identifier equal or greater than the value k.
It benefits of some good properties:

- **Load balance,** if the hash function is quite good this protocol can naturally balance the load between the nodes in a proper way.

- **Decentralization,** means that chord is distributed, so no node is more important than the others, with an improvement in robustness.

- **Flexible naming,** there are no constraint on the key structure, so it is very flexible for mapping applications' names to Chord's keys.

- **Scalability,** in fact the cost of the lookup of a key (number of hops that i have to perform for finding a key in the chord ring) grows as the log of the number of node present in the space address ($log(n)$). This is true if we use a scalable key location. In a scalable key location scenario each node in the space address has a smart selected number of information of the others nodes. If again m is the number of bits, each nodes maintain a table, called finger table, with m rows, each rows contain the info of the nodes that follow such node by at least 2^{i-1}, where i goes from 1 to m. For making a numerical example if m = 3 the finger table of a sample node N10 is composed using the following method: at first it has three rows, since m is equal to three; after we compute the information (identifiers) of the others friends nodes (lets suppose that we have others nodes, N12, N18, N20) in this way: first row = $10 + 2^{1-1} = 11$ (stored on N12), the second one = $10 + 2^{2-1} = 12$ (stored on N12), the third and last one = $10 + 2^{3-1} = 14$ (stored on N18), where 10 is the identifier of the node and the results are the values (the keys) for which N10 knows in which others nodes they are stored. The first row always has a key stored in the immediate successor. So at the end the finger table of N10 is the one below:

N10's Finger Table	
keys	Node Id
$10 + 2^{1-1} = 11$	Stored in N12
$10 + 2^{2-1} = 12$	Stored in N12
$10 + 2^{3-1} = 14$	Stored in N18

In this way a node n1 cannot (usually) find directly the other node n3 responsible for storing the keys k, but can find a friend node n2 that is close the the correct one reducing the number of needed hops (the algorithm converge to a complexity of $log(n)$). Another good scalability feature is that when a new node n2 is introduced between n1 and n3 the reorganization of stored keys is only local, for instance in our case we move from n3 only keys that fall in the range (n1...n2], since for these values the successor now is n2 and no more n3.

- **Availability,** in fact the Chord algorithm automatically updates the finger table of the various nodes when a new node is added in the ring, but also when a node fail, ensuring that (except for important failure) the node that store a key can be always found.

3.2.1 Chord's Fault-tolerance

Since our interest is to understand how fault-tolerance approaches works in a large scale distributed system, in this sub section we can focus our attention on what are the fault-tolerant method and guarantees offered by the Chord algorithm in a distributed peer to peer architecture when some node in the ring fails. But, before showing this process, it is better to know how the so called stabilization protocol works, so what happens when a new node joins the system and how the finger tables, of the nodes already present in the ring, is updated.
Suppose node n2 joins the system, and its ID lies between nodes n1 and n3. At first, n2 acquires n3 as its successor. Node n3, when notified by n2, acquires n2 as its

predecessor. When n1 next runs a specific stabilize command, it asks n3 for its predecessor, which is now n2, follows that n1 acquires n2 as its successor. Finally, n1 notifies n2, and n2 acquires n1 as its predecessor. At this point, all predecessor and successor pointers are correct. At each step in the process, n3 is reachable from n1 using successor pointers; this means that lookups are not disrupted.

Now, we analyze a situation in which nodes could fail and the correctness invariant that each node knows its successor can be broken. We try to understand in which way fault-tolerance is guaranteed in the chord algorithm to continue to have correct lookups. In fact, without any fault tolerance guarantee, when nodes fail simultaneously a node n could not be able to perform correct lookup for some key values anymore. For this reason, to increase robustness again faults, each node in the ring maintains also a list of r successors (following the clockwise order in the ring). In this way even if some nodes that are successors of n fail, n is able to direct a right lookup result. Only when all the successors node in the r list of n fail we have problems, but this is very less probable, and in addition increasing the size of r we decrease more this probability.

Having a successor list requires also some management in terms of stabilization protocol, but this can be done fortunately with few amendments of the process. Now a node n1 append the successors list of its successor node n2 to the last entry of its successors list. If n2 fails, n1 replace it with the first available entry (for example the one that identify the node n3) in its successors list and append to the last entry the successors list of n3 (the new available successor). When we perform lookup operations we search not only in the finger tables but also in the successors lists for retrieving the closest node to the correct one that stored some value. If a node fails during the lookup operation it proceeds, after a timeout, by trying the next best predecessor among nodes in the finger table and successors list. With this procedure we can still maintain a good performance in the lookup operations but also a good number of success in retrieving a key.

The last fault-tolerant guarantee of Chord is showed when a node n leaves the Chord ring voluntary, as if it were a failure of that node. This algorithm is also able to handle a situation like this. We can imagine that the usual node n2 is located between n1 and n3 and it wants to leave the space address. N2 will inform n3 that is going to leave and it will send to n3 its predecessor, so n3 will know that its predecessor is no more n2 but n1. In the same way n2 will inform n1 that is going to leave and it will send to n1 its successor, so n1 will know that its successor is no more n2 but n3 (n1 now can also update its successor list removing n2 and adding n3).

3.3 Final Comparison between Paxos and Chord

At first we need to say that both Paxos and Chord work for a large scale distributed system. But they have some differences, among which those relative to the implementation (the first one uses a complex message passing system for solving consensus problem, while the second one uses a distributed consistent hashing method for enabling lookups operation inside a system), but, more important, those relative to how fault tolerance works in such scenarios (ways for recovering after failure of nodes during the procedures by which both these method are divided), widely discussed in the previous sections of this chapter.
The first approach that we have discussed is very suitable for a master-slave architecture, using single-leader protocol in which one server is the most important one with respect to the others and it has the power of "making decision". In fact it works well when a single node, that acts as proposer, proposes a value to a set of others nodes that act as acceptors. It can also be used for re-electing a new leader if the previous one fails. This algorithm, in terms of performance is not so good since using lots of messages between agents can be very expensive, in fact Paxos suffer from the end-to-end message delays, and when fails occur, more messages are needed to be exchanged slowing the entire process of consensus. The message traffic of the Paxos can be regarded as the complexity of N, not considering Byzantine failure, otherwise the complexity could even be of (N^2). Finally studies have proved that the mechanism by which Paxos is based cannot tolerate more than 50% of the nodes crashing,

but still remain a good trade off between consistency and availability.
The Chord algorithm, on the contrary, is very adapt for a p2p architecture in which all the nodes play the same role inside the system (there isn't a master). The very nice property of this approach is that is very scalable, more than Paxos, when we add new nodes in the system, but it can also maintain good performance when a node fails, since the complexity of the algorithm still remain $(log(n))$. The only real failure that can compromise this protocol is the Byzantine one bu exist an extension of this method that allows you to tolerate such kind of faults. It is very strong to failure, again more than Paxos, in fact, considering also redundancy of replicas of nodes, when 50% of nodes fail, only 1.2% of lookups fail.

Difference between Paxos and Chord		
Aspect	Paxos	Chord
Approach	Message-passing	Consistent hashing
Reference Architecture	Master-slave	P2P
Complexity	N	log(N)
Fault tolerance guarantee	More than 50% can crash	When 50% of nodes fail, only 1.2% of lookups fail

4 Fault Tolerance Management in NoSQL Systems

In this last chapter of the article we will going to see what are the fault tolerance's approaches in different NoSQL systems. Specifically we will explain, based on chapters and articles found online, how the fault tolerance is implemented in three widely used systems:

- **Neo4j**, a graph based NoSQL management system.
- **MongoDB**, a document based NoSQL management system.
- **Cassandra**, a column based NoSQL management system.

4.1 Neo4j

Figure 5: Neo4j

Neo4j is an open source java-based scalable graph database management system, based on a master-slave architecture. It follows the property graph model. A property graph is a direct multi-graph where each node and edge can be associated to a properties (pairs of key value). every node can also be associated to one or more labels, while an edge can be associated to a type. This model is a schema-less model in the sense that we can add freely new edges and nodes with different structure.

Neo4j uses a specific query language called Cypher, that allows you to write query as path traversal in a graph for finding the result as sub-graph of the original graph. Some good properties of this graph database are the strong (casual) consistency, a quite high availability, a support for ACID transactions and the fact that is very scalable with respect to the number of nodes in the system, since the query execution is independent from this number, in fact the query execution time remains constant even if we add new nodes and edges (this property is called index-free adjacency).

4.1.1 System's Fault Tolerance Approach

Searching in the literature there are conflicting opinions of what fault tolerant protocol Neo4j uses. In fact, looking at the official documentation ([Neo]) of such NoSQL system we can discover that the Raft consensus protocol is used for leader election if some nodes (identified as the leader) fails. Other reference like [FP16] says that the multi-Paxos consensus protocol is implemented in Neo4j infrastructure to provide fault tolerance against node failures. For this reason we can explain in a simple way both the approaches that Neo4j could use for providing fault tolerance.

- **Raft protocol**. Raft was designed for better understandability of how consensus can be achieved considering that its predecessor, the Paxos algorithm, is very difficult to understand and implement. At first, during Raft algorithm each node in a replicated state machine (server cluster) can stay in any of the three states, leader, candidate and follower ([Geeb]). The rules of the interaction between these three different categories are: only a leader can interact with the client; any request to the follower node is redirected to the leader node; A candidate can ask for votes to become the leader; A follower only responds to candidate(s) or the leader; The leader regularly informs the followers of its existence by sending a heartbeat message; Each follower has a timeout in which it expects the heartbeat from the leader. The timeout is reset when the heartbeat is received; If no heartbeat is received the follower changes its status to candidate and starts a leader election.

The consensus problem is decomposed in Raft into two relatively independent sub-problems ([Wikc]):

- *Leader election*, when a leader fails a new one must be elected and a new term starts. A term is a period of time in which the leader election will occur. A server becames a candidate when it receives no messages from the leader during a period of time called election timeout and so now it starts the leader election process increasing the term counter, voting for itself as new leader, and sending a message to all other servers requesting their votes. If a candidate C1 receives in answer a term number greater than its term, from another candidate server C2, C1 loses the election and it recognizes the other candidate C2 as leader. Instead, if the candidate C1 receives the majority of votes by the followers, it becomes the leader. If neither of these two options happen the election starts again with a new term number.
- *Log replication*, at this moment the leader is the server responsible for log replication. This means that it can accepts the client requests and propagates them to the follower nodes, indefinitely, until they eventually be stored by the followers. Once that the leader receives from the majority of followers the confirmation that the log is replicated and stored it commits in its local state machine the operation, for guaranteeing consistency. If the log crashes there could be an inconsistency situation in which the new leader compares its log with the one of each follower, finding the last entry where they agree. Now the leader deletes all the entries coming after this critical point in the follower log and replace them with its own log entries.

Raft protocol ensures election safety, one single leader will be elected at the end of the procedure, even if followers crash, guaranteeing an eventually condition, that such failures are handled by the servers trying indefinitely to reach the downed follower.

- **Multi-Paxos protocol.** The execution of Multi Paxos protocol is very useful for a distributed system. It wants to implement a way for executing different instance of basic Paxos skipping, if the server is stable, the sub-phase 1 of the selecting value procedure, explained in the subsubsection 3.1.1 of this article. In this way the protocol reduces the failure-free message delay and also the overhead in communication ([oR]).

  - *Initialization:* at the beginning an instance of the basic Paxos is executed, in which both the phase 1 and phase 2 of selection step (election of leader) are executed.
  - *Further Execution:* for the next execution, instead of performing again the phase 1, in which a new proposer proposes value (proposes itself as the new leader), we can pass directly to the phase 2, the accept request stage, since the leadership is "sticky". In this way the number n associated to the proposed value remain the same.

  We don't have to worry about the worst case where leadership isn't stable (or distinct), because the algorithm will degrade gracefully into the general Paxos algorithm. So we ensure the same level of fault tolerance as Paxos, but in a very more efficient way.

In addition to these two methods, there are also others good practices used by Neo4j for increasing fault tolerance inside the overall system.

- The first one is the use of graph data replication among all the servers to provide redundancy in case of failures.
- The second practice is to maintain a quorum in order to execute write operations. With this requirement a minimum of servers need to be online for propagating write loads. The quorum allows the system to be reliable and available when nodes failures occur.

4.2 MongoDB

Figure 6: MongoDB

MongoDB is a general purpose, document-based, distributed database. MongoDB is a database that stores data as documents and supports general CRUD operations on one or many documents with a rich query language. Each document is a binary JSON-like object (called BSON format). In MongoDB we can have many databases, each database is composed by collections, which are similar to tables in a SQL database. Each collection is in turn formed by some documents that are divided in many fields (simple or complex fields). Documents are identified by unique ids, automatically created by the system. The main architectural features are the data sharding/partitioning among the replicas using a certain attribute identified as partition key. This partition can be the input for an hash function, if the approach to partition the data is the hash based one, or it can be the input of a condition, if we are using a range based partitioning. To manage the replication, ensuring consistency and a certain level of availability, it uses a master-slave approach.

4.2.1 System's Fault Tolerance Approach

Generally we can say that replication in MongoDB provides fault tolerance. In fact, replication allows to bring down a server if it is needed, for example for tasks like operating system patching, MongoDB patching or replace a hardware continuing to keep the entire database available. In MongoDB there are different fault tolerance levels that we can obtain, depending on different budget and business factors. ([Sol]): In order to achieve the fault tolerance's level 1 we need 3 host, if one of them

Fault tolerance's level in MongoDB		
Members	Majority required	F.T.'s level
3	2	1
4	3	1
5	3	2
6	4	2

goes down we have the others two that can elect a new primary (master) node. For obtaining fault tolerance level 2 we need instead 5 nodes.
In MongoDB, as the paper [SZ21] demonstrates, the object of replication is called oplog, a sequence of log entries, that is a single database operation stored in a dedicated oplog collection (a typical MongoDB's collection of documents). In this collection the oldest entries are deleted when they became useless, while the new ones are added inside the table. An oplog entry need to be replicated at least to a majority of servers to be a committed entry, so an entry that persist when a minority failure occurs. After replication, all servers of the same replica set will have identical oplogs. MongoDB's fault tolerance and leader (primary) election follows the Raft consensus protocol, already presented in the previously subsubsection 4.1.1 of Neo4j's fault tolerance approach, but, for this distributed database, the aforementioned protocol assumes some important differences and extensions in the implementation([Med]):

- **First difference**: in the basic Raft algorithm, before commits changes, the leader informs all the followers of these changes. In MongoDB, instead, the leader informs with an acknowledgment that the change is been written in one of the nodes and, starting from this node, the updates are forward propagated to the other nodes (followers). According to the previous digression about oplog, in the Raft-based systems a replica waits until the log entries are committed and then applies the log entries, while MongoDB introduces an optimization that speculatively applies an oplog entry when it is added to the oplog table and after commits the log entries.

- **Second difference**: instead of using a push approach in MongoDB we apply a pull based approach. In a push based method is always the primary nodes that can initialize a remote procedure call directed to the followers, for example for propagating writes updates to them. In MongoDB the primary wait for a follower to pull the new entries and, after that, the follower will propagate the changes to the other near secondary nodes and so on and so for, following a chain propagation. The main advantage is the performance reason, in fact, the replication becomes very fast and the leader's amount of work is not huge as before. An other benefit of the pull-based approach is that it enables a more flexible control of how data is transmitted over the network. Depending on users' needs, the data transmission can be in a star topology, a chaining topology, or a hybrid one.

- **Third difference**: in the Raft protocol the election timeout is chosen randomly and sometimes if the network band-with is slow we can have a lot of leader election. With MongoDB the user can select a personalized timeout, with a very nice flexibility.

- **Fourth difference**: in MongoDB we can also assign priorities to the nodes. If a node has an high priority this means that probably will be elected as leader, while, if it has priority 0 means that, this server will never run for election. Usually you can assign high priority when you know that a server is particularly strong with respect to the others.

- **Fifth difference**: in most of consensus system as Raft, before discarding old oplog entries the system take a snapshot of the relative database. In this way when a new server wants to join the system it can takes a look to the snapshot for retrieving the most up to date entries in the oplog collection (initial sync). MongoDB doesn't use snapshots approach and so a new server is added to the system it first chooses a sync source (another server) used for all the duration of the initial sync process. Now the new node save the timestamp of the most recent oplog of the sync source and start to clone the database. During this cloning process some inconsistency may happens, for instance some other nodes can update entries while the cloning is in progress, for this reason the final step is to remove inconsistency. If the process fails the new server start it again choosing another sync source.

- **Sixth difference**: Usually after the fail of a primary node the uncommitted oplog entries are lost. This issue, in MongoDB, can be avoided adding a new phase after a certain node wins the election and became leader. In this new phase, called catching phase, the leader cannot accept immediately new write requests. It has to wait until in its sync source (used since it is the new leader) a new entries is inserted, or after a certain user-defined timeouts. In this way we can preserve a lots of uncommitted writes.

- **Seventh difference:** in MongoDB there are more roles that a replica (follower node) can assume, along with the usual ones. A replica node in fact can be a arbiter. An arbiter is like a secondary node with the same power in voting the new leader, but it does not store any data. Another new role assumed by a secondary node is the non-voting member. Usually a node that is targeted as non-voting is a node that is associated with an intensive analytical workload and that store a lot of data, hence that it is not suitable for fault tolerance. In fact they only stores data but they cannot contribute to the voting procedure for electing the new primary node.

4.3 Cassandra

Figure 7: Cassandra

Cassandra is a column-based distributed database written in Java. Cassandra uses the CQL (SQL-like) query language that allows to perform the basic CRUD operations. Cassandra system is composed by keyspaces (like database in RDBMS). Each keyspace is formed by tables (or collection), that in turn are divided in rows and columns. Cassandra is a peer to peer architecture, with a leader less replication protocol. An important aspect of Cassandra is that it uses a distributed consistent hashing approach for what concern the partition of the data. The consistent hashing is the method used by Chord, discussed in subsection 3.2 of this paper. Just a little reminder: this approach wants to assign to a single space address, viewed as a ring, both the data and the nodes that store these data, using some defined partition key as input of an hash function. After this method provides for a look-up algorithm that converges to a good complexity of $log(n)$, for retrieving any data in the address space.

4.3.1 System's Fault Tolerance Approach

In general, the systems, like Cassandra, using distributed hash table, a structured P2P storage system, work very well in terms of availability, scalability and fault tolerance. In Cassandra some of the best practice for ensuring fault tolerance, as explained in the documentation of such NoSQL distributed system [Cas], are:

- **Replication strategies.** Cassandra replicates every data among the nodes of a cluster according to a specific replication strategy. Specifically each key-value pairs is replicated to multiple nodes (replicas) to ensure fault tolerance and availability ([GSG21]). Each replication strategy has a replication factor: a number indicating in how many other nodes a data is duplicated. Usually, replication strategies allows to skip nodes that are in the same failure domain, for instances that can be found in the same rack. In doing so Cassandra can tolerate failure of a whole rack. The two usable replication strategies are NetworkTopologyStrategy and SimpleStrategy. For each Cassandra's keyspace (database) we can use a different strategy.
  - *NetworkTopologyStrategy*, this first method requires a specified replication factor for each data-center. If the number of racks is greater than or equal to the replication factor for the data-center, each replica is guaranteed to be chosen from a different rack, otherwise, each rack will hold at least one replica, but some racks may hold more than one.
  - *SimpleStrategy*, in this second strategy a single number as replication factor can be defined, all nodes are treated equally, ignoring any configured racks.
- **Probabilistic Quorum System and Consistency Level.** A probabilistic quorum system requires each client to receive responses from a certain number of nodes W, after a write operation, or from a certain number of nodes R, after a read operation. For instance the strong consistency requirement is satisfied in Cassandra when the chosen value for W and R respect the following expression: $R + W > n$ (n is the number of nodes). So we can use these two numbers for tuning the different consistency level of Cassandra, depending on our needs. For example when we are in a situation in which the previous expression is: $R + W <= n$ the

system can only ensures eventual consistency, probably to improve throughput and latency.

- **Gossip Protocol.** The previous information for choosing quorum and for replicating nodes are based on the assumptions that we know what are the nodes that are alive and the ones that are dead. This knowledge derive from a failure detection mechanism based on a gossip protocol ([Edu]), very suitable for a peer to peer architecture, especially because it can reduces the amount of chaotic communication between nodes. in Cassandra's gossip protocol, nodes share information not only about themselves, but also about some other nodes they know. When one node talks to another, the node which is expected to respond not only provides information about its status, but also provides information about the nodes that it had communicated with before. The main feature of this approach is to ensure that every nod will respond with most recent information and that we can define the ring membership of each node.

- **Failure Detection (Phi Accrual).** Gossip protocol create the basis of ring membership but is the failure detector that can discover when a node is working or not. Every node in Cassandra runs a version of the Phi Accrual failure detector. In this way, a node can understand if their peer nodes are alive or not, based on one main aspect: the state of received heartbeats (periodic messages send among peers nodes in a P2P architecture). If the node doesn't receive increasing value in the heartbeat messages from another peer nodes, after a certain amount of time, the failure detector marks such peer node as dead and the system will not route request to it anymore. If, or when, the nodes will start to send updated heartbeat messages again, Cassandra tries to establish a connection with such node, marking it as available. As we can see Cassandra never removes a node from the system if it is not explicitly defined, even if it is marked as down, because, otherwise, would be necessary to perform a lot of re-balance inside the ring, also because the node could restart again.

5 Conclusion

At the end of this paper we have understood what is fault tolerance in general and what are the main related issues and the main approaches, digging into the details also of three NoSQL distributed database systems.

We can point out that also the consensus problem is strictly related to fault tolerance, since most of the protocol (like the Paxos or Raft protocol) are used for a new leader election after the crash of the previous one. For briefly summarize the main difference between the two analyzed method for fault tolerance, the Paxos protocol and the Chord algorithm, we can say that the first one is: suitable for master slave architecture in a single leader scenario, it is very complex in terms of implementation, even if is able to manage a good number of faulty nodes. It suffers also of communication congestion. For this reason I have also explained another version of this method when we were talking about the Neo4j fault tolerance's approaches (subsubsection 4.1.1), the multi-Paxos protocol. Another consensus used in fault tolerance, that derives from the Paxos one, is the Raft protocol, again implemented in Neo4j for election after a leader's crash. Also MongoDB uses Raft for some fault tolerance aspect but with some important differences due to the specific aspect of Mongo, first among all, the fact that it is pull based. The second discussed algorithm (Chord) is instead very adapt in a peer to peer architecture, so for leader less protocol. It works very well in term of performance and it is very scalable when we add or remove, due to failure, nodes in the system. Cassandra uses this algorithm for data partition and also fault tolerance, together with other good fault tolerant practices, like replication, probabilistic quorum system, for tuning the different consistency level, gossip protocol and the associated Phi Accrual failure detector.

In conclusion we are able to say that fault tolerance is a needed features for most of the situation in a large scale distributed system, for dealing with data loss, network fault, leader crash and node crash in general, with procedure for finding the failure and with method for managing these failures, sometimes complex, but necessary for

the data survival, important aspect for the user point of view bu also for a company point of view.
As future works I suggest to myself, but also to the most curious and brave readers, to delve into some advanced topic: approaches for optimizing the fault tolerant process in such distributed systems. For doing that we can read accurately the [GSG21] paper about an implementation for improving Cassandra's fault tolerance system called Cassandra+, or the [SZ21] paper regarding an efficient pull based consensus approach in MongoDB.

References

[Cas] Apache Cassandra. Dynamo | apache cassandra documentation. `https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html#incremental-scale-out-on-commodity-hardware`.

[Edu] Edureka. Gossip protocol in cassandra. `https://www.edureka.co/blog/gossip-protocol-in-cassandra/`.

[FP16] Beniamino Di Martino. Florin Pop, Joanna Kołodziej. *Resource Management for Big Data Platforms.*, pages 189–203. Springer, 2016.

[Geea] GeeksforGeeks. Recovery in distributed system. `https://www.geeksforgeeks.org/recovery-in-distributed-systems/`.

[Geeb] Parikshit Hooda. GeeksforGeeks. Raft consensus algorithm. `https://www.geeksforgeeks.org/raft-consensus-algorithm/`.

[GSG21] Juan Mantica Haochen Pan Darius Russell Kish Lewis Tseng Zezhi Wang Yingjian Wu. Guo-Shu Gau, Kishori Konwar. Practical experience report: Cassandra+: Trading-off consistency, latency, and fault-tolerance in cassandra. *International Conference on Distributed Computing and Networking.*, 2021.

[Imp] Imperva. What is fault tolerance. `https://www.imperva.com/learn/availability/fault-tolerance/`.

[IS03] David Liben-Nowellz David R. Kargerz M. Frans Kaashoekz Frank Dabekz Hari Balakrishnan. Ion Stoicay, Robert Morrisz. Chord: A scalable peer-to-peer lookup protocol for internet applications. 2003.

[Kle17] Martin Kleppmann. *Designing Data-Intensive Applications.*, pages 278–319. O'Reilly, 1st edition, 2017.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News.*, 2001.

[Med] Medium. Raft consensus algorithm and leader election in mongodb vs coachroachdb. https://medium.com/geekculture/raft-consensus-algorithm-and-leader-election-in-mongodb-vs-coachroachdb-19b7

[Neo] Neo4j. Leadership, routing and load balancing. https://neo4j.com/docs/operations-manual/current/clustering/setup/routing/.

[oR] Amber Feng. Amber on Rails. Paxos/multi-paxos algorithm.). https://amberonrails.com/paxosmulti-paxos-algorithm.

[Sol] Raul Salas. Mobile Monitoring Solution. Mongodb replication and fault tolerance. https://mobilemonitoringsolutions.com/mongodb-replication-fault-tolerance/.

[SZ21] Shuai Mu. Siyuan Zhou. Fault-tolerant replication with pull-based consensus in mongodb. *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation.*, 2021.

[Wika] Wikipedia. Heisenbug. https://en.wikipedia.org/wiki/Heisenbug.

[Wikb] Wikipedia. Paxos (computer science). https://en.wikipedia.org/wiki/Paxos_(computer_science).

[Wikc] Wikipedia. Raft (algorithm). https://en.wikipedia.org/wiki/Raft_(algorithm).