Distributed Algorithms: Short Survey Paper*

Omar Jarkas
School of Information Technology and Electrical Engineering
University of Queensland
Brisbane, Australia
o.jarkas@uqconnect.edu.au

***Abstract*—The world's growing reliance on mission-critical applications, ranging from avionics to blockchain, demands reliable distributed systems. With the explosion of different interconnectivity and technologies; software bugs, human error, and malicious attacks are the norm rather than the exception. With such failures, fault-tolerant algorithms are essential. Fault-tolerant distributed algorithms allow a collection of machines to survive and recover from failure; all the while reaching agreement and working as a single cohesive system. However, while such technology gave birth to cryptocurrencies and facilitated the reliance on the cloud today, they are hard to understand, design, and implement. This paper provides a brief introduction to the distributed algorithm. It covers some environment and properties assumptions while giving a taxonomy of the major distributed algorithms, their use cases, and variants.**

***Index Terms*—Fault-Tolerance, Consensus, Atomic Commit, Partially Synchronous, Reliable Broadcast, Byzantine Agreement, Paxos, Raft.**

I. Introduction

Distributed algorithms are one of the technologies powering reliable systems [1]. From medical and military equipment to avionics to blockchain, distributed algorithms are crucial for environments where faults are unacceptable. The world today is heavily reliant on distributed systems. However, like any technology, distributed systems can be vulnerable and prone to failure. For that, fault-tolerant systems use replication and redundancy. The intuition behind such a reliable system is the reliance on the exact same replica in case one or more fails. However, replication is not enough. For a replicated service to handle faults and continue running simultaneously, it requires a clever algorithm that enables collaboration among the nodes involved and abstracts such replication so as to convey one coherent logical process to the client [2], [3]. Such algorithms are called distributed algorithms. While the goal is the same, fault-tolerance, the purpose of such algorithms vary. From getting different independent nodes to agree on a single value, to ensuring messages are delivered in the presence of faults to database transaction replication, distributed algorithms are at the core of such replication.

However, while the concept behind fault tolerance distributed algorithm is straightforward, designing fault-tolerant distributed algorithms are not. The primary reason behind that is that algorithms have to run in partial states. Due to replication, distributed algorithms operate in an environment where there is no one point of failure. This decentralization of nodes implies partial awareness of the global system in each node. Such partial awareness makes distributed algorithms hard to debug and reason about. Moreover, distributed algorithms serve different goals and environments. Often such diversity may lead to varying levels of environment and system assumptions and types of failure which often must be dealt with the avoid unforeseen results [4], [5].

For systems to achieve such fault-tolerance, the consensus is the main problem to solve in distributed algorithm [3]. Consensus is the process of ensuring independent nodes agree on the state of the system in the presence of faults. This is to ensure one cohesive system. Generally, the consensus is used for state machine replication. Any system that changes it state-based in a certain input can be thought of as a state machine [6]. Thus the job of a distributed algorithm is to replicate it unto the different participating nodes hence State Machine Replication (SMR). State Machine Replication is discussed more in section II-E. Distributed algorithms' main goal is to achieve state machine replication via consensus.

Over the years various types of consensus protocols have been proposed to solve different goals. In this paper, we survey different fault-tolerant and consensus protocols. The algorithms are classified into three categories based on their end goal. While Consensus is the term used by all distributed algorithms to achieve agreement and fault-tolerance, the paper classifies algorithms into three types based on their end goal. Consensus algorithms are presented and discussed separately in the context of State Machine Replication SMR. The three types are Reliable Broadcast for message delivery, Atomic Commit for distributed database replication, and Consensus for reaching agreements.

GBai: 1. Problem to address: why this survey is needed? What gap can it fill?
2. Introduction of this survey: on what aspects? how (categorization, classification criteria, etc.)? goals?
3. Contribution of this survey

The contribution on this work is that it provides a survey, classification, and taxonomy on different types of distributed algorithm and there relation to consensus. The work provide basic understanding of distributed algorithm by explaining different concepts behind them. Moreover, the worh surveys and explores such algorithms along with there properties and significance.

The paper is divided as follows. Section 2 covers the dif-

ferent environment assumptions and properties of distributed systems, along with some definitions and system specifications. Section 3 classifies distributed algorithms into classes based on their end goal. Section 4 goes into the discussion before concluding in Section 5.

II. Preliminaries

As mentioned, distributed algorithms serve varying end goals and environments. This section goes over the different environment considerations, assumptions, and properties of distributed algorithms. The section starts by introducing the Byzantine Generals problem before presenting different environment assumptions, types of faults, and group membership.

A. Byzantine Generals Problem

The Byzantine General Problem named was derived from a thought experiment problem in which Generals of an Army need to agree on whether to attack the city Byzantium or treat in the presence of traitorous general [2]. The problem emphasizes the need for a majority agreement to achieve consensus. Of course in distributed systems there are no generals and army, however, such a problem can be model in distributed systems to achieve fault-tolerance via consensus. Over the year many variants of such a problem have been proposed to solve consensus [7], [8], [4], [9], [10], [11], each under different assumptions and settings.

B. Synchronous vs Asynchronous Systems

Synchronous distributed systems are predictable systems. Such systems come with a high number of environment guarantees that makes designing and reasoning about distributed algorithm easier [12]. Controlled environments such as cloud systems can implement constraints that facilitate synchronous algorithms. In such environments, information such as processing speed and message delivery are known. With such knowledge, assumptions on the upper bound of message delivery are possible. Messages not received in such time or order constraints have a high probability of being faulty. The presence of such assumptions significantly aids distributed algorithm design.

Dwork et. al [13] defines the upper bound as Δ which represents the time latency between nodes or Φ which represents the relative speed of each process running the consensus module on each node. Moreover, some systems might have the privilege of having a synchronized clock that can enable failure detection and facilitate ordering. Such system constraint can potentially achieve total synchronization since it has the ability to force nodes to execute in lockstep synchrony. Lockstep synchrony is the process of executing processes of independent nodes at the exact same time, such capability can be achieved in strongly synchronized systems that have Δ, Φ, and Globally Synchronized Clocks (GSC).

While synchronous systems provide many advantages and can achieve a high degree of consensus and fault-tolerance, they are not practical due to their strong system assumptions. Strong assumption means that systems must always be predictable in order to work. Such predictions are not always present which impacts such algorithms' robustness and applicability in environments where such assumptions are unknown or unreliable. More suitable to real-world scenarios are asynchronous systems at don't hold Δ, Φ, GSC assumptions. In asynchronous environments, messages can be delayed unpredictably while having out-of-sync clocks with no fixed upper bound of Δ and Φ. An example of such an environment is the internet. Unfortunately, purely asynchronous distributed algorithms are very hard to design when there are no bound assumptions Δ and Φ. This is because it is impossible to determine whether a process is faulty or merely takes an infinite amount of time. In fact, Fisher et. al [14] proved that it is impossible to reach consensus in a pure asynchronous environment where only one node can fail. This is the well-known FLP impossibility which states that a simple problem has no deterministic solution in an asynchronous system in the presence of one simple failure [15].

While the consensus problem cannot be solved in a purely asynchronous environment, there have been many workaround and assumptions to reach consensus in an asynchronous environment. Mainly the use of partial synchrony [14]. Partial synchrony utilized the assumption of the presence of a special event (Global Stabilization Time (GST)) to find a middle ground between synchrony and asynchrony. This synchrony also assumes an unknown yet finite upper bound on time Δ in which the environment synchrony alternates before and after GST. Message are considered if they arrive before $\Delta + max(x, GST)$ as an upper bound.

C. Types of failures

As mentioned, different environments demand different assumptions. From having control over it to performance tradeoffs, members configuration, and security, all play a role in the design and the implemented distributed algorithm. One of the assumptions a system needs to figure out is related to failure assumption. The type of failure possible and the effect on the system can affect the design of the system. Generally, fault-tolerance distributed algorithms are classified based on the type of error they can handle. According to [16], the most common types of faults found in a system are categorized into three types. Most algorithms are considered fault-tolerant if they follow the Byzantine model of handling $n >= 3m+1$, where m is the number of faulty processors and n is the total number [9].

1) Fail-stop: The safest type of failure. In a fail-stop environment, the effect of errors and failures are minimized. In such an environment, processes are either correct or have crashed [16]. The idea behind such failure is that it is either detectable and can be halted or the failure is benign and only affects the termination of it one process [17]. A fail-stop is considered in some literature as a crash [4]. A system is said to be k-crash tolerant if it can survive up to k fail-stops [4].

2) Byzantine: Byzantine failures, originally coined after the Byzantine Agreement Problem [2], where traitorous generals attempt to make the army fail by sending false messages, are

one of the most disruptive. Byzantine failures are any that can cause a process to act in an arbitrary manner affecting other nodes. From hardware and network failure to cyberattacks, Byzantine failures allow the process to diverge from the distributed protocol by sending false information. Nodes in the system have no way to distinguish whether the information is correct. Hence, Byzantine failure can cause more harm to the system than fail-stop. The Byzantine Agreement problem centers around such kinds of failures [18]. Protocols that can tolerate up to t processes that exhibit Byzantine failures are said to be t-Byzantine resilient [4].

3) Transient: Similar to Byzantine, transient failure can achieve the same damage and can occur for a range of reasons [16]. However, the key difference for such failures is the time and the solution. Transient failure unlike byzantine is time-bound in which a failure can occur for a short time and can be debugged and solved by the system with no downtime. Such systems that handle transient failure are often dubbed as self-stabilizing or self-healing systems [16].

D. Group Membership

In a distributed system, nodes can be scattered in various ways. One key aspect that goes into designing a distributed algorithms is node connectivity and membership configuration. Such assumptions can contribute to strong assumption constraints (synchronous) or weak ones (asynchronous). While such assumptions are deterministic and don't go into the core of consensus and fault-tolerance algorithms they do pose a unique set of challenges. Environments such as blockchain must handle admission of any new nodes, on the other hand, some cloud quorum of nodes is restricted by strong assumptions of admission and consolidation, hence admission can be hard.

1) Open Group: Such systems are mainly for asynchronous systems where there are weak assumptions. Like Bitcoin, in an open group distributed environment any node can be integrated into the system [19]. For that, a bootstrap process must be included in the distributed algorithm to allow such integration. Such environments are considered one of the hardest to maintain due to the lack of sanitization i.e. malicious nodes can join to harm the system. Often an incentive mechanism is required to bribe joining members to adhere to correctness. An Example of such incentive algorithm are Proof-of-Work (PoW) and Proof-of-Stake (PoS) found in Bitcoin and Ethereum respectively. While such systems allow access to any new node, it is still assumed that there is message authentication in which each node can reliably determine the source of the message sender [4].

2) Closed Group: For closed groups, nodes must make sure that the communication is authenticated and fit the assumption constraint of the system [19]. For such authentication, a mechanism to managing node membership has to be built-in as not to face trust issues [19]. For small systems, such authentication can be relatively straightforward.

E. State Machine Replication (SMR)

In general, a consensus algorithm is typically referred to in the context of a replicated state machine. Nodes generals attempt to reach consensus in order to replicate a state machine on different nodes. Given a set of nodes running a certain service, how can we make sure that the state of each node doesn't diverge from each other? Lamport showed that SMR are generallt the way to achieve consensus [20]. An example of a service is a state machine that handles transactions request in a cryptocurrency. State machine replication is the problem of devising an algorithm to get all the connected nodes to agree on the next block to maintain a coherence system and state for all the nodes running the service. Other examples of state machine replication are discussed in Section III-C with Paxos and Raft consensus algorithms.

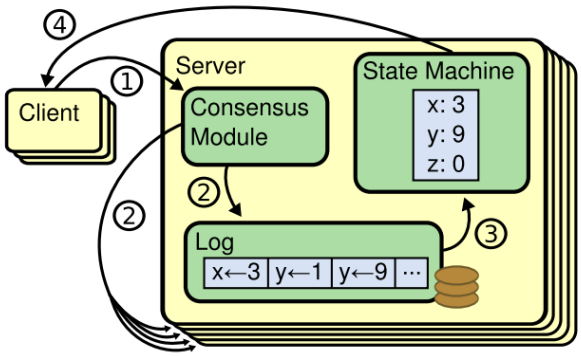

Fig. 1. SMR Diagram taken from [21]

III. Distributed Algorithms

As mention in I, in this paper, distributed algorithms are classified into 3 types based on what they are trying to achieve.

Consensus is a fundamental problem in which fault-tolerance is achieved [22]. While consensus can be a part of algorithms classified as part of Reliable Broadcast or Atomic Commit, consensus algorithms are discussed separately. In the Reliable Broadcast section, the goal of the included algorithms is to ensure messages are delivered to all nodes in the presence of failure. On the other hand, the main goal behind Atomic Commit has distributed database replication. While all such sections include voting and agreement techniques to reach consensus, consensus algorithms in this section are regarded in the context of state machine replication.

GBai: 1. How are these algorithms categorized, based on goals, target problems, or what? Basically, how are categories different from each other?
2. How to make sure the categories are complete

A. Reliable Broadcast

GBai: 1. In each category, are there the root and variants? What differentiate them?
2. are there a general model/work flow/architecture for each family?

One of the first and simplest problems solved for the fault-tolerant distributed algorithm. The problem of guaranteeing that a message is received by all alive nodes in the presence of failure was solved by a seminal [5]. There are many variants of such an algorithm. However, all of them must agree to the same basic validity, agreement, and integrity properties. Over the years after reliable broadcast there has been work exploring such algorithms and variants of it under different consideration [23], [24], [25].

For most of these algorithms, the dominant assumption is the nodes can crash only in a fail-safe manner and the channels have perfect communication links. While there are many variants of such algorithms under different reliability considerations, the most well-known of them are Best-Effort Broadcast, Regular Reliable Broadcast, and Uniform Reliable Broadcast. The key idea behind such algorithms is a condition which states that if a message is delivered to a node by a non-faulty node, the exact message is also delivered by all other no-faulty nodes to the same node in the system. This allows nodes to evaluate the correctness of the message received by accepting the message that is most received i.e. the majority of the nodes sent. However, such algorithms differ with the number of correct processes necessary for reliability depending on system assumption and tolerance appetite.

Moreover, some algorithms must ensure total order i.e. messages must be received in the order they are sent in. Such algorithms are primarily used for state machine and database replication in which order in everything [26], [27]. Such algorithm is also called Atomic broadcast [27]. Over the years there have been many algorithms introduce that cater to different properties, assumptions, and objectives [28].

B. Atomic Commit

Atomic commit protocols (ACP) are a special type of Byzantine Generals problem [29] that are mainly used for atomic database transactions replication [30]. In distributed database systems, transactions must be replicated in many sites and divided into sub-transactions, however, due to failures that can occur, databases require extra measures to ensure consistency across different sites [18]. Otherwise inconsistent data can be stored permanently [31]. Similar to SMR consensus algorithms, distributed commit algorithm's role is to ensure synchronization among replicas of transactions in a distributed database such that data can be updated in the presence of failure to guarantee global unanimous outcome [18], [29].

Such algorithms can vary in safety, reliability, and efficiency depending on different assumptions [32]. In terms of reliability, failure in such algorithms is assumed as fail-safe i.e. originator of the transaction can fail however not maliciously. Moreover, such algorithms are considered safer than other since database site must agree on one value, either to commit or not rather than a state. Another assumption for reliability is the eventual recovery of nodes in which all failed nodes are set to recover eventually [29]. For efficiency, aspects such as message complexity, number of nodes, and recovery mechanism play a big role [29]. On the other hand, such replication and consensus can be costly in terms of time [33]. Due to many algorithms having many message exchanges, latency, wait time, and the number of phases can cause performance bottlenecks. In section III-B2, different commit algorithms are surveyed that attempt to deal with different assumptions and optimizations such as performance and reliability.

First introduced in 1979 [31] as a crash recovery system to guarantee reliable storage, atomic commit algorithms have evolved to suit many assumptions and environments [29]. The paper introduces the concept of atomic transactions. A transaction is any piece of information that has atomic property. Atomic property is the all-or-nothing property. The following is an example of a global transaction taken from [34] in which each t can be treated as a sub-transaction.

- t_1 Order a ticket at Northwest Airlines;
- t_2 Order a ticket at United Airline;
- if t_1 fails;
- t_3 Rent a car at Hertz;
- t_4 Reserve a room at Hilton;
- if t_5 fails;
- t_1 Reserve a room at Sheraton;
- t_1 Reserve a room at Ramada;
- if t_4 and t_5 fails;

In order to ensure atomic commit, a two-step approach is generally required [31]. These steps are the voting and decision which are considered as phases [29]. The first step is concerned with pre-committing the necessary information in an intention set without updating the data stored yet. In this stage, each transaction has a coordinator (typically the originator of the transaction) which will initiate voting of all nodes on whether to commit or not. After getting a majority or unanimous vote (depending on the algorithm) from all the sites involved, the second phase starts. The second phase is involved with each site committing the transaction to its local storage. Only after committing the change, are they written in one step into the permanent storage. If a failure happens it can only happen between such phases since the phase will be automatically restarted if an error happens until the transaction is consistent in all storage sites [31]. Such algorithms are dubbed synchronous since the recovery and voting process are done via timeouts. Two-phase commit is one of the most standard database synchronization algorithms [29].

1) Recovery: According to [29], in a distributed database, there are four places where communication failure might occur.

a) Participant - Phase one: Before voting, when the sites are waiting for a pre-commit message in order to vote, any of the participants can fail and abort the transaction. Upon its recovery, the participants scan its logs for transactions without a corresponding final decision. After the coordinator sends the transaction the participant reply with acknowledgment Ack. Only upon receiving Ack can the coordinator forget the participant that aborts. This ensures the participants get updated once they recover. Variants of 2PC are optimized to

accommodate such final decision even in the rare case when the coordinator fails and can't remember the participants [30].

b) Coordinator - Phase one: The coordinator can fail while receiving votes from participants. Since the coordinator is still in phase one and didn't commit yet, the coordinator aborts the entire transaction. When the coordination is back up, a series of steps are taken to commit pending transactions by rebuilding the protocol table lost upon failure. The protocol table is a local table on the site of the coordinator containing all pending transactions. Such transactions are the ones in the queue waiting to be replicated and unfinished ones that got aborted due to failure. After resending the pre-commit for each of the transactions, participants abort the previous pre-commit for this transaction and vote in the new attempt.

c) Coordinator - Phase Two: During phase two the coordinator can fail. The participant can finish phase one and send their votes to the coordinator waiting to commit. However, since the coordinator has crashed, participants don't receive a final decision commit or abort. In such a case, the participant will wait for the decision indefinitely or timeout. Upon coordinator recovery, the coordinator scans the log for pending transactions. Following, it re-submitting the pre-commit and waits for the new vote. The participant in this case can either revote on the transaction or send an Ack and commit the transaction.

d) Participant - Phase Two: During phase two, participants can fail after voting and before committing. If the coordinator didn't receive Acks from all participants it resubmitted the final decision to the initial unacknowledged nodes. If no reply is received from the node by the coordinator, the coordinator remembers them until recovery. When nodes recovery they enquire about the transaction, commit it, and sends Ack.

2) Algorithms: As mentioned, the Two-Phase commit is one of the first algorithms that solved atomic commits consensus in distributed databases. However, over the years there have been many different algorithms proposed to solve such synchronization under different assumptions [35], [30], [36]. In addition, two common variants of Two-Phase commit were introduced to optimize performance and reliability by reducing message traffic and log writes [30]. Those variants are Presumed Abort (PA) and Presumed Commit (PC). Moreover, one algorithm that favors reliability in the reliability-performance trade-off in Three-Phase commit [35].

To start, PA is similar to 2PC, however, it attempts the following concepts to reduce message exchanged that also solve rare 2PC flaws. PA follows a no-information abort rule in which transactions are assumed to be aborted if recovered nodes don't have outcomes with the coordinator. This reduces Acks for abortion processes and the forced log write necessary to remember aborts. On the other hand, PC follows an opposite rule in which only commits are treated as the norm and are not acknowledged, only upon process abortion are the Acks sent and log updated. Finally, 3PC solves the blocking of participants in case of coordinator failure by implementing an extra phase named the pre-commit-phase. The Pre-commit phase allows a global decision to be made where additional logs are used to handle coordinator failure.

PA and PC solve different struggles of 2PC, other environment-dependent algorithms have been proposed over the years. Moreover, such algorithms can guarantee performance optimization and fewer phases. Starting with an algorithm that addresses main memory databases' atomic commit problem in one phase [36]. Main memory databases or MMDB are a database with local storage residing in main memory instead of on the disk. This protocol drops voting between cohorts. The coordinator sends the final decision to the participant and remembers unacknowledged nodes to remember and replicate the committed transactions once they recover. The author claim that the proposed protocol achieves total atomicity and durability by reducing communication by a third to achieve better performance.

Anther protocol the promises high-performance optimization is proposed by Haritsa et. al [33]. They propose and evaluate a new atomic commit protocol named PROMPT (Permits Reading Of Modified Prepared-data for Timeliness). This protocol was designed with performance in mind to suit high-performance and real-time systems. They claim high performance via reducing data inaccessibility and priority inversion by a concept called "optimistic borrowing". The concepts allow sites to update their state temporarily. According to [29], [33], [36], performance in such algorithms is measured by three factors. Firstly, the degree to which the protocol affects the execution and processing of transactions i.e. Effect of Protocol on Normal Processing time. The second performance metric is the tendency of nodes to fail, how likely are nodes to fail, or their resilience to failures. Lastly, one of the key performance factors is the site recovery speed after crashing. The PROMPT algorithms attempt to tackle performance issues related to the resiliency and the effect of such protocols on the processing time. It is mainly captured due to the concept of lending discussed earlier in which cohorts are allowed to take uncommitted data before the protocol is done. This allows them to continue their process, if required, then resynchronize once there done. Other variant include optimization of Two-Phase Commit protocol in web, mobile database, and commercial distributed environment, these are [37], [38], and [39] respectively.

C. Consensus

Reliability is the main goal behind distributed algorithms[7]. It is often thought off in terms of consensus. The consensus problem was first introduced as a problem between generals in which they must agree whether to attack or retreat [2], hence byzantine generals problem. Various variants of Byzantine Agreements have been proposed over the years that take into account the number of participants, the number of rounds, performance trade-offs, and goals of the algorithm [40], [10], [8], [11].

In the context of reliable computing, a consensus is merely the process of enabling a set of nodes or processes to agree. Generally in such models, a consensus is reached via a

majority voting [2], [4]. However, in the presence of faults, depending on the system, voting may not be enough to agree in one step since faulty processes can hinder agreement by sending conflicting votes [4]. For that, a multi-stage voting scheme is proposed to overcome the problem [41].

Building on The Byzantine Agreement, Fischer [4] surveyed different types of Byzantine General agreements under different assumptions. The paper evaluated different authentication and synchrony conditions of the Byzantine Generals' problem.

While Byzantine Agreement solved the problem of consensus under adversarial conditions, there was still a need for a verified protocol that can achieves consensus in asynchronous environments. Paxos was published introduced in 1998 by Leslie Lamport [42], to this day, it is considered one of the most influential works done on consensus in which many algorithms propose forward were based on it. The algorithm is a 2 phase voting algorithms that guarantee consensus in the present and faults. The algorithm was designed to tolerate fail-stop failures [43].

1) Paxos:

a) How Paxos works: As mentioned, there are very variants of Paxos, each with its unique properties and optimizations. Such algorithms are discussed in the following chapters. One of the things all such variants have in common is that they are all based on Basic Paxos, Single decree Paxos, or Classic Paxos. The goal of Paxos is to achieve state machine replication via consensus [44]. The entire algorithm along with its properties and assumptions are explained in [6]. Basic Paxos is a two-phase protocol made up of Proposers and Acceptors. The protocol's only aim is the get nodes to agree on one value such that only one value is ever chosen. Proposers are any node that are required to propose a new value and Acceptors are nodes that attempt to agree on such proposed value. A Proposer can be an Acceptor simultaneously. Basic Paxos guarantees the following properties:

- at most one value is chosen (safety)
- a server only learns the chosen value (safety)
- one of the proposed value is eventually chosen (liveness)
- all nodes eventually learn about the chosen value (liveness)
- all nodes eventually learn about the chosen value (liveness)

In Paxos 2 phases are necessary since the first phase is about accepting the value (accepting doesn't guarantee agreement) and the second phase is about the agreement from the majority voting. However, some obstacles might appear in such 2 phase algorithm. If all proposed values are accepted, the algorithm might end up with more than one majority. Paxos ensures safety by following the following conditions. The first condition is if a value has been chosen by the majority of the nodes, no new node can propose a new value. Figure III-C2b demonstrates the condition. The figure shows how two majorities can occur if such a condition is not implemented.

Moreover, such a condition alone doesn't yet guarantee not ending up with multiple majorities. The second condition states that if the value has been chosen, any competing proposal must be aborted. Such property can be ensured by proposal order. Figure III-C2b demonstrates how two majorities are possible amongst 5 servers if such a condition is not followed.

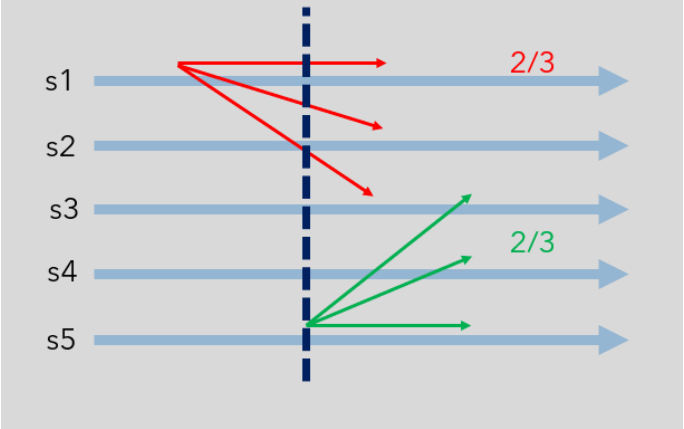

Fig. 2. Basic Paxos First Condition.

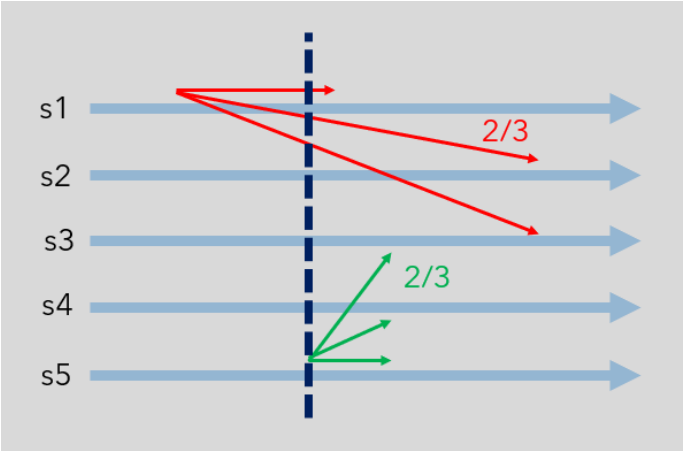

Fig. 3. Basic Paxos Second Condition

b) Paxos Over the Years: As mentioned, Paxos is the most influential consensus algorithm. This section will cover different variants of Paxos. Starting from Paxos variants in asynchronous environments to synchronous, leaderless, and Byzantine variants. The section also discusses the implementation of Paxos in highly available systems.

In 2004, Cheap Paxos was introduced [44]. Cheap Paxos is a variant of Paxos that replaces participating nodes with auxiliary ones that only handle failure. The authors claim that such a variant of Paxos is cheaper, hence Cheap Paxos. It utilizes cheap auxiliary nodes to handle failure by taking part in reconfiguring the system and removing failed processors. By having a 3rd of the nodes handling failures, such auxiliary nodes can be slow, small, and cheap. The author guarantees consistency properties where only two-thirds of the nodes participate in consensus. However, the paper fails to guarantee liveness. This is since, in Dynamic Paxos where the state ma-

chine determines the set of acceptors, nodes cannot guarantee progress if all the chosen acceptors crash. The paper makes the distinction between the type of Paxos, Static and Dynamic Paxos. It claims that traditional Paxos fall under Static while Cheap is Dynamic.

After Cheap Paxos in 2005, Lamport introduced the Generalized Paxos Algorithm [22]. The paper introduces a variant of Paxos that attempts to agree on more than one value in an incremental order i.e the next value must be larger than the current. The author argues that efficiency is increased in this generalized Paxos algorithm by allowing concurrently issued commands to be executed in two message delays. The paper outlines the consistency, liveness, stability, and non-triviality property that the algorithm satisfies.

Following generalized Paxos, Fast Paxos, a simple extension of Paxos was published [45]. In 2006, it came as an optimization to Basic Paxos in which two message delays are possible to achieve consensus instead of three in the case where no competing proposals collide. In Paxos there might be the case that two or more proposers propose a value concurrently, this is coined collision proposal. In Fast Paxos, message delays are reduced by bypassing the leader and sending proposals directly to the acceptors. One of the conditions of Paxos is that it used that 3 message delays to ensure that there are not simulate majority and conflict. The paper proposes a property to avoid collision by always considering the majority a quorum where there can't be more than 1 quorum without intersection, thus ensuring majority voting and reducing message delays. Both Fast and Generalize Paxos offer optimization for Basic Paxos.

c) Paxos Implementation: While Classic Paxos sufficiently solve consensus, it is merely an asynchronous proof of consensus that requires many additional assumptions to handle the fault in a distributed systems [46]. Such a protocol doesn't have any assumption on time nor does it allow continuous agreement of nodes periodically. For that many implementation of Paxos has been proposed subsequently.

The first two implementations of Paxos were in 1996 [20], [47]. Lampson [20] was one of the first to acknowledge the Paxos implementation gap for highly available systems and proposed a general implementation of Paxos. The protocol attempts to utilize Paxos to replicate a deterministic state machine on different nodes.

Following that, in 2003, Gafni et. al implemented Disk Paxos which is a variant of Basic Paxos for a system of processors and disks [48]. Another implementation of SMR was presented by Lee et. al [47]. This time the implementation was to achieve fault-tolerance via consensus in replicating distributed virtual disks. By incorporating Paxos to achieve automatic replication the author claims to achieve many features of the ideal storage system since it provides high availability, accessibility, performance, and no management.

In [49], the authors attempt to solve replication and fault-tolerance concerns for storage infrastructure. They use and implement Paxos and other concurrency safety primitives to achieve replication, consensus, and atomicity. Such an algorithm can be thought of in the context of atomic commit [36].

In 2007 and 2009 an attempt to build a simple implementation of Paxos was published in the form of Muti-Paxos [46], [50]. Multi-Paxos is an attempt to implement Paxos in a fault-tolerant database system with minimal possible enrichment. The author of the paper claims that the simplest way to implement Paxos in scalable systems in which agreement can be reached for log replication is via periodically implementing Paxos every replication procedure as instances, hence Multi-Paxos. This such protocol is considered synchronous due to upper bounds necessary for instances and timeout, however, partially synchrony is possible for log replication. The paper attempts to optimize Paxos by reducing the number of packets required to reach consensus. Moreover, it proposes a solution for recovery and lagging replicas. Multi-Paxos is the most widely deployed implementation of Paxos [51].

In 2016, M^2Paxos, which is an implementation of Generalize Paxos discussed earlier, was introduced.

d) Mutli-Leader Paxos: One of the biggest drawbacks of Paxos is the performance overhead in synchronous systems [51]. As mentioned Paxos is a consensus algorithm that agrees via a majority vote, however, for Paxos to work, a leader must be elected, which is demonstrated in [6], [46] While having one leader eliminates contention and the unforeseen consequences of concurrency, it has major drawbacks in terms of performance. In large systems, such as WANs containing a large number of nodes, reaching consensus becomes a linearly exhausting task.

To solve such a problem, work has been proposed that implements Paxos with multi or no leader [52], [53], [54]. First introduced as a patent in 2007 by Lamport et. al [55], the multi-leader protocol ensures that different leaders are assigned a different number in the sequence of replicated commands. Thus ensuring a unified order of command in state machine replication and avoiding contention. Based on such work Mencius was proposed in 2008 [53]. Based on a partitioned leader scheme the main goal of this protocol is the increased throughput via partitioning the consensus protocol among the nodes in the server. The algorithm addresses contention and load balancing concerns.

To start EPaxos is a new distributed algorithm to achieves load balance and enables the system to evenly distributed the load across all replicas [52]. EPaxos replaces the leader in the Paxos by allowing clients to choose what node to communicate with. It can achieve consensus by implementing a dynamic decentralized ordering constraint on commands issued, such ordering enables non-faulty processes to reach consensus on the order of command independently.

Shifting from basic Paxos multi-leader to generalized consensus Paxos, Alvin is an implementation of leaderless Paxos in Go [54]. Alvin is a combination of the EPoxas, Maninus, and generalize consensus. In essence, it is the combination of their advantages for building a more optimized and scalable multi-leader Paxos in a geographically replicated transactional system.

e) Byzantine Paxos: Paxos and all its variants and implementation discussed prior are fail-stop; they don't handle

Byzantine Failures. This section covers Paxos variants for Byzantine failures. In a paper published in 2001, Lampson discusses the different variants of Paxos in their appetite for faults. In it, the paper compares a derivative of Paxos for Byzantine faults and discusses its safety, liveness, and performance [56]. The paper goes on to compare fail-stop and byzantine failures in Paxos in which they can each tolerate $\frac{n-1}{2}$ and $\frac{n-1}{3}$ faults respectively.

Following such a paper by a year, Liskov et. al [57] proposed the Practical Byzantine Fault Tolerant algorithm based on SMR using Paxos. Moreover, Byzantine Paxos was again covered in [43]. In both cases, the algorithms proposed can tolerate up to $\frac{1}{3}$ failure at once. Finally, a byzantine leaderless algorithm was discussed for basic Paxos by Lamport [58].

2) Raft:

a) Rafts Advantages: One of the major drawbacks of Paxos is it is notoriously hard to understand and implement in real-life scenarios. Also, Paxos doesn't provide a good foundation for practical implementation. Since the implementation of single-decree Paxos and multi-Paxos are not clearly outlined and there are many gaps in the architecture and assumptions, often it takes developers a lot to understand and when trying to build often they end up building a variant unproved protocol. Raft, came as an alternative to Paxos where understandability and replication are a big part of the design [21].

First, raft decomposes the consensus algorithms into 3 steps, each of which is treated as independently as possible, clearly explained, and examined. The first step is leader election, after that is the mechanism in which raft ensures agreement, and the third step is ensuring safety and liveness properties during SMR. The second thing that raft does differently is eliminating non-determinism wherever possible. Leaders are elected deterministically following safety property.

b) Leader Election: In any window of time, a node is one of those three states. A Leader, which there can only be one at a time. The leader is elected via all the nodes amongst a group of candidates each term. The election is a simple first-request-first-vote algorithm with the term going to the new majority leader. Once the new leader is known by all nodes, the leader handles all client's requests and is responsible for replication and maintenance of log consistency across all nodes. Nodes communicate via RPC for voting and replication. If a node does receive a message in a specified timeframe from the leader it assumes the leader has crashed and starts another appointment, so the leader establishes his authority via Heartbleed Empty Append RPC. In the presence of a split vote, direct reelection will start.

c) Log Replication: Once a leader is elected, it is the one that communicates with the client, once the client sends data (for example a transaction) the leader first appends the entry into his logs and attempt to maintain consistency across all nodes by replicating his log. Since we are dealing with possible faulty processes, inconsistencies can appear due to a series of leader-node crashes. A leader may be elected to find inconsistency in many nodes, it is his job to ensure the all replicated log are consistent with his log. To achieve such

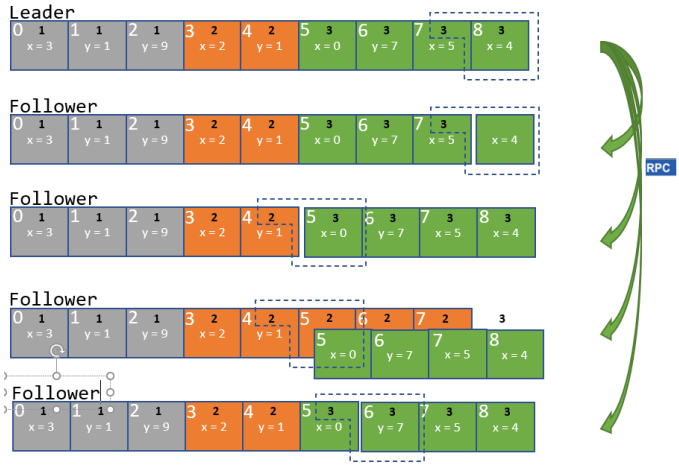

Fig. 4. Raft Replication Algorithm

replication the leader compares his previous index and current term with each inconsistent node, if the term or index are different the leader attempts to find a time where both logs were consistent and over the remaining with his log.

d) State Machine Safety: Now going back to state machine replication, while leader election and replication are a crucial part of the consensus algorithm, the soul of the algorithm is ensuring that each state machine executes the same order of commands. For that, there is some restriction on which nodes may be elected as leaders. This is a safety property. For example, a follower might be unavailable while the leader commits several log entries, then it could be elected leader when it comes back online and overwrites these entries with new ones (trying to be consistent); as a result, different state machines might execute different command sequences. For that, a Leader Completeness property is baked into the algorithm in which no leader is elected unless he has all the log committed with the latest commit (the commit is the last state where all online nodes are consistent).

[?] Algorithm	Environment	Leader	Failure
Basic Paxos	Asynchronous	Yes	Fail-Stop
Cheap Paxos	Asynchronous	Yes	Fail-Stop
Generalized Paxos	Asynchronous	Yes	Fail-Stop
Multi-Paxos	Synchronous	Yes	Fail-Stop
Disk Paxos	Asynchronous	Yes	Fail-Stop
Boxwood	Synchronous	Yes	Fail-Stop
EPaxos	Asynchronous	No	Fail-Stop
Mencius	Asynchronous	No	Fail-Stop
Alvin	Asynchronous	No	Fail-Stop
PBFT	Asynchronous	Yes	Byzantine
Byzantine Paxos	Asynchronous	Yes	Byzantine
Byzantine Leaderless Paxos	Synchronous	No	Byzantine
Raft	Partial	Yes	Fail-Stop

TABLE I
TAXONOMY OF DIFFERENT SMR CONSENSUS ALGORITHMS

Table III-C2d shows a taxonomy of the different algorithms discussed in this section.

GBai: Suggest a section on the insights concluded from the survey and open questions for future research

IV. Discussion

Over the years different distributed algorithms were introduces to solve and optimize different problems. Even though distributed algorithm face many challenges in terms of design, implementation, performance, and scalability, they are increase getting more and more popular die to the contribution they bring. Distributed Algorithm help relieve many critical application from faults and make them more secure and attack-resistant. Since systems are not created equal, and reliable can come in my forms, distributed algorithm came to serve different goals with limited capability. Since then, many variant of such algorithms contributed to performance and implementation. This paper is a way to demonstrate the significance and evolution of such algorithms without deciding which is the best overall.

V. Conclusion

This is a survey paper that cover different types of fault-tolerant distributed algorithms. The paper cover perimilary details about the nature, setting, and envrioments of such algorithms. Afterward, the paper attempts to classify algorithms based on their end-goal. The classes are Reliable Broadcast, Atmoic Commit, and Consensus. Enevthough consensus is mainly required in all of them to achieve fault tolerance, consensus is treated as a seperate class and define in the context of State Machine Replication. The paper defines the classes of algorithms before proceed to survey all the there varaint and examine them under different setting.

References

1 Konnov, I., Lazić, M., Veith, H., and Widder, J., "A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 719–734.

2 Lamport, L., Shostak, R., and Pease, M., "The byzantine generals problem acm transactions on progamming languages and syetems, vol. 4 no. 3 pp. 382-401," 1982.

3 Van Steen, M. and Tanenbaum, A., "Distributed systems principles and paradigms," *Network*, vol. 2, p. 28, 2002.

4 Fischer, M. J., "The consensus problem in unreliable distributed systems (a brief survey)," in *International conference on fundamentals of computation theory*. Springer, 1983, pp. 127–140.

5 Srikanth, T. and Toueg, S., "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distributed Computing*, vol. 2, no. 2, pp. 80–94, 1987.

6 Lamport, L. *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

7 Garcia Molina, H., Pittelli, F., and Davidson, S., "Applications of byzantine agreement in database systems," *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 1, pp. 27–47, 1986.

8 Lynch, N. A., Fischer, M. J., and Fowler, R., "A simple and efficient byzantine generals algorithm." GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, Tech. Rep., 1982.

9 Pease, M., Shostak, R., and Lamport, L., "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

10 Dolev, D. and Strong, H. R., "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

11 Dolev, D., Reischuk, R., and Strong, H. R., "Early stopping in byzantine agreement," *Journal of the ACM (JACM)*, vol. 37, no. 4, pp. 720–741, 1990.

12 Stoilkovska, I., Konnov, I., Widder, J., and Zuleger, F., "Verifying safety of synchronous fault-tolerant algorithms by bounded model checking," in *TACAS 2019-International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

13 Dwork, C., Lynch, N., and Stockmeyer, L., "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

14 Fischer, M. J., Lynch, N. A., and Paterson, M. S., "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

15 Mostéfaoui, A., Mourgaya, E., Parvédy, P. R., and Raynal, M., "Evaluating the condition-based approach to solve consensus," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* IEEE Computer Society, 2003, pp. 541–541.

16 Fisman, D., Kupferman, O., and Lustig, Y., "On verifying fault tolerance of distributed protocols," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 315–331.

17 Schlichting, R. D. and Schneider, F. B., "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 3, pp. 222–238, 1983.

18 Mohan, C., Strong, R., and Finkelstein, S., "Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 89–103.

19 van Steen, M. and Tanenbaum, A. S., "A brief introduction to distributed systems," *Computing*, vol. 98, no. 10, pp. 967–1009, 2016.

20 Lampson, B. W., "How to build a highly available system using consensus," in *Distributed Algorithms*, Babaoğlu, Ö. and Marzullo, K., Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–17.

21 Ongaro, D. and Ousterhout, J., "In search of an understandable consensus algorithm," in *2014, Annual Technical Conference*, 2014, pp. 305–319.

22 Lamport, L., "Generalized consensus and paxos," 2005.

23 Chang, J.-M. and Maxemchuk, N. F., "Reliable broadcast protocols," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 3, pp. 251–273, 1984.

24 Awerbuch, B. and Event, S., "Reliable broadcast protocols in unreliable networks," *Networks*, vol. 16, no. 4, pp. 381–396, 1986.

25 Kaashoek, M. F., Tanenbaum, A. S., and Hummel, S. F., "An efficient reliable broadcast protocol," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 4, p. 5–19, Oct. 1989. [Online]. Available: https://doi.org/10.1145/70730.70732

26 Lamport, L., "The implementation of reliable distributed multiprocess systems," *Computer Networks (1976)*, vol. 2, no. 2, pp. 95–114, 1978.

27 Agrawal, D., Alonso, G., El Abbadi, A., and Stanoi, I., "Exploiting atomic broadcast in replicated databases," in *European Conference on Parallel Processing*. Springer, 1997, pp. 496–503.

28 Défago, X., Schiper, A., and Urbán, P., "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

29 Al-Houmaily, Y. J. and Samaras, G., "Two-phase commit." 2009.

30 Mohan, C., Lindsay, B., and Obermarck, R., "Transaction management in the r* distributed database management system," *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 4, pp. 378–396, 1986.

31 Lampson, B. and Sturgis, H. E., "Crash recovery in a distributed data storage system," 1979.

32 Gray, J. N., "Notes on data base operating systems," in *Operating Systems*. Springer, 1978, pp. 393–481.

33 Haritsa, J., Ramamritham, K., and Gupta, R., "The prompt real-time commit protocol," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 2, pp. 160–181, 2000.

34 Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M., "A multidatabase transaction model for interbase," 1990.

35 Skeen, D., "Nonblocking commit protocols," in *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, 1981, pp. 133–142.

36 Lee, I. and Yeom, H. Y., "A single phase distributed commit protocol for main memory database systems," in *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE, 2002, pp. 8–pp.

37 Yu, W. and Pu, C., "A dynamic two-phase commit protocol for adaptive composite services," *International Journal of Web Services Research (IJWSR)*, vol. 4, no. 1, pp. 80–100, 2007.

38 Nouali1&2, N., Drias, H., and Doucet, A., "A mobility-aware two-phase commit protocol," 2006.

39 Samaras, G., Britton, K., Citron, A., and Mohan, C., "Two-phase commit optimizations in a commercial distributed environment," *Distributed and Parallel Databases*, vol. 3, no. 4, pp. 325–360, 1995.

40 Dolev, D., Fischer, M. J., Fowler, R., Lynch, N., and Strong, R., *An efficient Byzantine agreement without authentication*. IBM Thomas J. Watson Research Division, 1982.

41 Davies, D. and Wakerly, J. F., "Synchronization and matching in redundant systems," *IEEE Transactions on Computers*, vol. 27, no. 06, pp. 531–539, 1978.

42 Lamport, L., "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

43 ——, "Byzantizing paxos by refinement," in *International Symposium on Distributed Computing*. Springer, 2011, pp. 211–224.

44 Lamport, L. and Massa, M., "Cheap paxos," in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 307–314.

45 Lamport, L., "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

46 Du, H. and Hilaire, D. J. S., "Multi-paxos: An implementation and evaluation," *Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02*, 2009.

47 Lee, E. K. and Thekkath, C. A., "Petal: Distributed virtual disks," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996, pp. 84–92.

48 Gafni, E. and Lamport, L., "Disk paxos," *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003.

49 MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L., "Boxwood: Abstractions as the foundation for storage infrastructure." in *OSDI*, vol. 4, 2004, pp. 8–8.

50 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*, 2007, pp. 398–407.

51 Peluso, S., Turcu, A., Palmieri, R., Losa, G., and Ravindran, B., "Making fast consensus generally faster," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 156–167.

52 Moraru, I., Andersen, D. G., and Kaminsky, M., "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372.

53 Barcelona, C.-S., "Mencius: building efficient replicated state machines for wans," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

54 Turcu, A., Peluso, S., Palmieri, R., and Ravindran, B., "Be general and don't give up consistency in geo-replicated transactional systems," in *International Conference on Principles of Distributed Systems*. Springer, 2014, pp. 33–48.

55 Lamport, L., Hydrie, A., and Achlioptas, D., "Multi-leader distributed system," Aug. 21 2007, uS Patent 7,260,611.

56 Lampson, B., "The abcd's of paxos," in *PODC*, vol. 1. Citeseer, 2001, p. 13.

57 Castro, M. and Liskov, B., "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

58 Lamport, L., "Brief announcement: Leaderless byzantine paxos," in *International Symposium on Distributed Computing*. Springer, 2011, pp. 141–142.