Fundamental Algorithms in Distributed Systems

Zhiliang Wan
United World College Changshu China, China.

* Corresponding author. Tel.: +86 13121099737; email: zlwan20@uwcchina.org
Manuscript submitted November 10, 2022; revised November 28, 2022; accepted December 10, 2022.
doi: 10.17706/jsw.18.1.44-54

Abstract: A distributed system is a collection of spatially separated processes that communicate over a network and coordinate their actions by exchanging messages. Because of the failure-prone nature of the network and the processes themselves, many complex problems arise in distributed systems. If not addressed, these problems can prove to be significantly costly to applications that involve communication and coordination between multiple processes. Many of these problems can be overcome through the use of fundamental algorithms for ordering, coordination, and agreement. This paper will review some of these algorithms, including synchronization, ordering of events, mutual exclusion, election, multicast, consensus, and Byzantine Generals Problem.

Keywords: Distributed systems, Synchronization, Fault tolerance, Agreement

1. Introduction

In a distributed system, multiple processes communicate over a network and coordinate their actions to achieve certain objectives. Before devising algorithms, one must identify a system model to specify assumptions of how the network and the processes behave [1].

The basic assumption about the underlying network is that processes communicate via bidirectional point-to-point channels. These channels can be assumed reliable, where a message is received if and only if it is sent. Alternatively, these channels can be assumed to be unreliable, where messages may be lost or duplicated. In general, communication channels in actual networks tend to be unreliable, but these unreliable channels can be made reliable if the sender continually retransmits lost messages and the receiver discards duplicated messages. Since unreliable channels can be made reliable, many of the algorithms reviewed in this paper assume reliable communication channels.

In respect to the behavior of the processes, it can be idealistically assumed that processes do not fail. In reality, however, it is important for algorithm designers to be pessimistic with their assumptions and consider unlikely but possible scenarios. The failure model of processes can be crash-stop, in which the processes stop executing forever after failing, or crash-recovery, in which the processes may resume executing after a finite amount of time. It can even be assumed that processes behave in arbitrary, potentially malicious manners, and algorithms have been devised to cope with such a failure model.

Under certain assumptions of the system model, researchers in the past decades have devised algorithms to solve various important problems in distributed systems, including synchronization, ordering of events, mutual exclusion, election, multicast, consensus, and Byzantine Generals Problem. These algorithms are very important to applications that rely on the underlying distributed system.

2. Time and Clocks

In a distributed system, it is ideal for every process to have relatively accurate knowledge about the physical time. This knowledge is important in various situations, which include determining timeouts, calculating some duration of time, and recording timestamps for events [1]. Because of the inherent limitations of the hardware clocks on ordinary computers, it is tremendously important for processes to obtain information of physical time from external sources that are more accurate. As it is impossible for processes to have completely accurate knowledge of physical time, logical clocks were invented to capture the ordering of events that occur on different processes [2].

2.1. Physical Clocks

Computers typically have hardware quartz clocks that oscillate at an almost constant rate [1]. By counting the number oscillations, a computer is able to obtain information about the number of seconds elapsed since a certain time point, which is usually the Unix epoch of January 1st, 1970. Unfortunately, under this method, the clock is subject to deviation from the actual time when the rate of quartz oscillation varies, perhaps due to changes in temperatures or other factors. As time elapses, the discrepancy between the clocks on separate computers gradually increases, which is known as clock drift. As a result, many computers tend to synchronize their clocks with more accurate external sources. These sources typically rely on atomic clocks to derive the Coordinated Universal Time (UTC) as a standard reference.

2.2. Network Time Protocol

The network time protocol (NTP) is selected for the review because it is the dominant algorithm that synchronizes the physical clock of processes in a distribute system. The network time protocol synchronizes the clocks of clients to the clock of a server, which relatively closer to UTC [3]. NTP divides the processes in a distributed system into strata i, where stratum 0 has an accurate UTC. A client on stratum i synchronizes with a server on stratum $i-1, i>0$. Essentially, the synchronization procedure involves a roundtrip of message exchange to estimate the one-way delay and the clock skew between the client and the server. The client can then mitigate this skew by adjusting the rate of its clock for a designated period of time.

However, there are two significant limitations of the network time protocol in its primitive form. Firstly, since the structure of the NTP closely resembles a tree rooted at the server on stratum 0, any failure of a server can jeopardize the synchronization of its descendants. Secondly, it is unsound to assume that each one-way delay is exactly half of the total roundtrip delay. This assumption fails when there's queueing in the network as well as other unforeseeable factors, leading to inaccuracy [3]. To mitigate these two limitations, a client can perform the protocol with multiple servers. This approach not only reduces the probability of all servers failing simultaneously but also improves accuracy by using average values to calculate the clock skew.

2.3. Happen-before Relation

The happen-before relation, denoted by $\rightarrow$, is selected for this review because it is of critical importance to the literature, laying a foundation for many later works on distributed systems. With an understanding of special relatively, Lamport realized that there is no invariant ordering of two events unless one event can casually affect the other event [2]. Therefore, he proposed the happen-before relation, which is independent from physical time, to capture the potential causality thus the ordering between events in a distributed system. The happen-before relation is defined by the following properties.

(1) If event a and event b occur in the same process, and a occurs before b in the order the code execution of the same process, then $a \rightarrow b$.

(2) If event a is the sending of a message and event b is the receiving of the message, then $a \rightarrow b$.

(3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

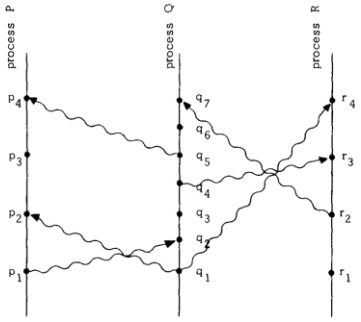

Fig. 1. Sequence diagram for the happen-before relation [2].

In Figure 1, since event p_1 occurs before event p_2 in the order of the code execution of process p, $p_1 \rightarrow p_2$ according to (1). Because p_1 is the sending of a message and q_2 is the receiving of the message, $p_1 \rightarrow q_2$ according to (2). Moreover, because $p_1 \rightarrow q_2$ and $q_2 \rightarrow q_4$, $p_1 \rightarrow q_4$ follows from the transitive property (3). The chain of events that follows from the transitive property (3) can be arbitrarily long. For example, the previously arrived conclusion $p_1 \rightarrow q_4$, combined with the observation that $q_4 \rightarrow r_3$ from (2), can lead to a further conclusion that $p_1 \rightarrow r_3$ using (3), and so on and so forth.

The significance of the happen-before relation lies in the notion of potential causality [2]. If $a \rightarrow b$, then a may or may not caused b. If neither $a \rightarrow b$ nor $b \rightarrow a$, then a and b are said to be concurrent, or $a \mid\mid b$, where no potential causality is involved in the relationship between the two events.

2.4. Lamport Clocks

The Lamport clock mechanism is included in this review because it is the first algorithm to determine the happen-before relation between events in a distributed system. This mechanism involves a software counter maintained by every process, whose value is independent of physical time [2]. Whenever an event occurs, the Lamport clock mechanism assigns the counter value to the event.

The timestamp of event e is denoted by $L(e)$, and L_i is the counter maintained by process p_i [3]. The Lamport clocks algorithm works as follows.

(1) Prior to the execution of a relevant event at process p_i, $L_i = L_i + 1$.

(2) When p_i sends a message, it piggybacks the value of L_i as t on the message.

(3) When p_i receives a message, $L_i = max(L_i, t)$

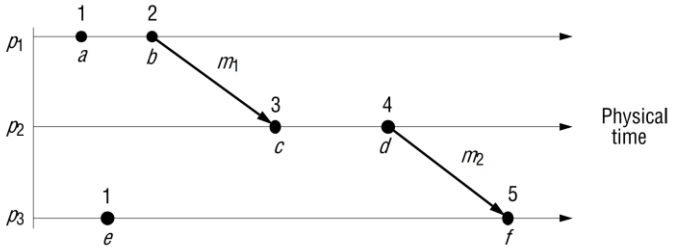

Fig. 2. Sequence diagram for Lamport clocks [3].

In Figure 2, since process p_1 increments its counter L_1 from 0 to 1 prior to event a according to (1), the process assigns its counter value 1 to event a. Because event b is the sending of a message and event c is the receiving of the message, the Lamport timestamp of c is greater than the timestamp of b by 1.

Although the value by which L_i increments is 1, any positive integer would suffice [2]. By induction, it can be shown that $a \rightarrow b \Rightarrow L(a) < L(b)$. In the above example, because $a \rightarrow b$, $L(a) > L(b)$. However, $L(a) <$

$L(b) \Rightarrow a \rightarrow b$ is false because if $L(a) < L(b)$, then either $a \rightarrow b$ or $a \,||\, b$ is true. In the above example, although $L(e) < L(b)$, $b \,||\, e$, which means $e \rightarrow b$ is false.

2.5. Vector Clocks

The vector clock mechanism is chosen for this review because it allows processes to determine the partial ordering between two events using the timestamps, overcoming a significant limitation of Lamport clocks [3]. In a system of N processes, every process p_i maintains a vector of N elements to timestamp local events. Initially, all elements of all vectors are zero. Similar to Lamport clocks, p_i increments the ith element of its vector prior to the execution of a relevant event and assigns its vector to the event. When p_i sends a message, it attaches its vector on the message. When p_j receives the message, if an element on the attached vector is greater than the corresponding element on the vector p_j maintains, p_j updates the element on its vector to the greater value.

Given event a and event b, if every element in vector of a is less or equal to the corresponding element in the vector of b, and a and b are different events, then a happens-before b [3]. This is because the vectors provide sufficient information to demonstrate that the events happened-before a constitute a subset of the events happened-before b. Therefore, the mechanism can determine the happen-before relation between two events by comparing their vectors, which is a significant improvement over Lamport clocks. In addition, vector clocks can also demonstrate if two events are concurrent by vector comparison. It is important to determine concurrency in many systems, such as a replicated database [1]. When multiple clients are updating values in multiple replicas, the application must know if these updating events are concurrent or not to resolve potential conflicts.

However, there are inherent limitations to vector clocks [3]. Firstly, it is challenging to apply vector clocks to dynamic systems where processes join and leave, as all processes must be able to insert and delete elements in their vectors. Secondly, vector clocks are expensive due to the $O(N)$ space complexity resulting from storing each vector and the $O(N)$ time complexity resulting from each comparison.

3. Coordination and Agreement

3.1. Mutual Exclusion

The problem of mutual exclusion arises when processes in a distributed system share a resource and only one process can enter the critical section at a time [3]. The critical section is a section of the program that accesses the shared resource, perhaps a variable or an object. To prevent conflicts, no more than one process can enter the critical section at a time. The simplest algorithm is one where a central server issues a token to permit a process to enter the critical section and retrieves the token when it leaves the critical section. If a process requests the token while another process is in the critical section, the server queues its request based on the order of message arrival. This algorithm guarantees that only one process enters the critical section at a time. However, the algorithm fails to achieve fairness because the ordering of the queue is based on physical time of message arrivals at the server instead of the happen-before order, making it possible for a process to enter the critical section multiple times while another process waits to enter. In addition, the system fails if the server fails, and the server may become a potential performance bottleneck.

Ricart and Agrawala's algorithm is selected from the literature because it is the first algorithm that solves mutual exclusion with a distributed approach. For a process to enter the critical section, it sends a request message to the other $N - 1$ processes and attaches a Lamport timestamp to the message. After receiving all $N - 1$ responses, the process enters the critical section. When receiving a request message, if the receiving process is currently in the critical section, or it is attempting to enter the critical section and has a timestamp-identifier pair smaller than that of the requesting process, it queues the requests according to

the comparison of the timestamp-identifier pair. Otherwise, the receiving process replies to the requesting process. After a process exits the critical section, it replies to the process at the front of the queue. The algorithm guarantees that only one process is in the critical section at a time because the process that is in the critical section will not reply to the requesting process until it exits the critical section. Fairness is ensured because the requests are ordered according to the Lamport timestamps with arbitrary identifiers breaking ties, which guarantees that a process only enters the critical section once while another process waits to enter. However, the $O(N)$ messages sent incur a significant overhead on the network and makes the algorithm impractical in large systems.

Maekawa devised an algorithm that only uses $O(\sqrt{N})$ messages to achieve mutual exclusion [5]. The voting set of a process consists of K processes, where $K \approx \sqrt{N}$. Maekawa's algorithm is chosen for the review because it is considered to be a significant improvement over the expensive Ricart and Agrawala's algorithm. For a process to enter the critical section, it sends a request message to every process in its voting set and enters the critical section after it has received all K votes. When a process receives a request message, if it is currently in the critical section, or if it has already sent a vote to a requesting process, it queues the requests according to the comparison of the timestamp-identifier pair. Otherwise, it replies to the request and sets its state to voted. When a process is informed that another process has exited the critical section, it replies to the request from the process at the head of the queue and sets its state to voted. Only one process can enter the critical section at a time because the processes at the intersection of voting sets can only vote for one process at a time. Requests eventually succeed and fairness is ensured because of the use of Lamport timestamps. Although this algorithm sends significantly less messages than the above one, its delay between one process exiting the critical section and the next subsequently entering is worse because two messages are required instead of one. Therefore, a tradeoff must be made by the system designer.

However, all of the above mutual exclusion algorithms do not tolerate unreliable message transmission and the failure of processes that assume certain roles [3]. Fault tolerance can be achieved through general consensus algorithms, which will be discussed afterwards.

3.2. Elections

Analogous to elections of political figures, distributed elections choose coordinators, perhaps as the servers in the centralized mutual exclusion algorithm [3]. Distributed election algorithms must choose a single coordinator instead of multiple. This is challenging when multiple processes initiate elections concurrently, as the algorithm must choose the process with the highest identifier among all N processes in the system with unique identifiers.

The ring-based election algorithm is chosen for this review because it was one of the first algorithms to solve the problem of distributed election. In the ring-based election algorithm, the neighbor of process p_i is process $p_{(i+1)modN}$ [3]. When a process initiates an election, it sends a message containing its identifier to its neighbor. When a process receives such a message, if the identifier in the message is greater than the identifier of the receiving process, it propagates the message to its neighbor. Otherwise, the process only propagates the message with its identifier if it is not currently participating in an election. If a process receives a message consisting of its identifier, it means that all processes in the ring have agreed that this process should be elected. Therefore, the process sends a message around the ring announcing its status as the new coordinator. Although the algorithm correctly selects the process with the greatest identifier while tolerating concurrent elections, it is not fault-tolerant because the failure of one process on the ring undermines the entire algorithm.

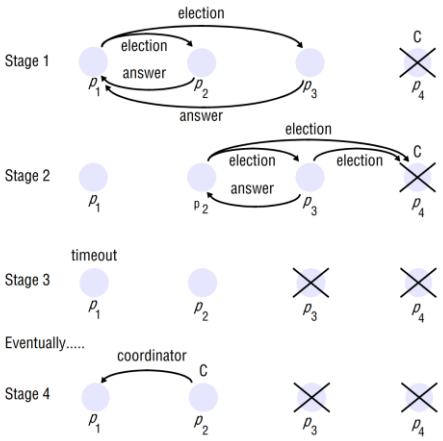

Fig. 3. Progress of the bully algorithm [3].

The bully algorithm is chosen from the literature because it was one of the first fault tolerant algorithms to elect the process with the highest identifier [3]. The bully algorithm is considered to be a significant improvement over the ring-based election algorithm because the former is fault tolerant whereas the latter is not. In the bully algorithm, all processes know which processes have higher identifiers. A process initiates an election after discovering the possibility of failure of the previous coordinator using timeouts. The process can simply elect itself if it knows it has the highest identifier. Otherwise, the process sends an election message to the processes with higher identifiers. If it receives no answer message after a timeout, it announces itself as the coordinator to the rest of the processes because assuming a maximum delay of message delivery, all processes with higher identifiers have failed. If the process receives any message, that means one or more processes with higher identifiers are still functional. These processes with higher identifiers will continue to execute the algorithm that the first process executed, eventually electing a functional process with the highest identifier. In Figure 3, process p_1 initiates an election due to speculating the failure of the previous coordinator p_4. p_1 sends an election message to both p_2 and p_3, who have higher identifiers. Upon receiving the messages, p_2 and p_3 continue to execute the same algorithm that p_1 executed, which is to send messages to processes with higher identifiers. Eventually, since process p_3 has failed, p_2 has not received an answer message from p_3 after a timeout. Therefore, p_2 announces to p_1 that it is the new coordinator. To prove that the algorithm guarantees only a single coordinator, assume the contrary that several coordinators were to coexist. Those with lower identifiers will not be coordinators because they will have received an answer from the process with the highest identifier, which contradicts the assumption and proves the case of a single coordinator. However, the algorithm relies on the assumption of a maximum delay of message delivery, which is impractical in real systems. Furthermore, the algorithm requires $O(N^2)$ messages if the process with the lowest identifier initiates an election, making the algorithm unrealistic for large systems. Therefore, consensus algorithms with weaker assumptions and greater efficiency must be devised, and these algorithms will be discussed afterwards.

3.3. Group Communication

Multicast refers to fault tolerant message transmission to multiple processes [6]. It is solely based on point-to-point communication between processes, without functionalities supported on the hardware level such as IP multicast. In the simplest multicast algorithm, a process sends individual messages to other processes, but this algorithm is unreliable because the processes will have inconsistent knowledge about the message if the sender fails before sending all the messages. Therefore, reliable multicast is designed in such a way that every process sends the message to the rest of the processes after delivering it, which

guarantees that the correct processes will either all deliver the message or all not deliver it. Since reliable multicast involves no timing assumptions, it tolerates an asynchronous system model. However, reliable multicast involves a total of $O(|g|^2)$ messages, where $|g|$ is the size of the group g involved in the multicast, making the algorithm inefficient and impractical for large systems.

The reliable multicast algorithm results in processes delivering multiple messages in arbitrary orders due to the unpredictable behavior of the underlying network. This is unacceptable in many scenarios, such as a replicated database system whose updates from the messages are not commutative [1]. Therefore, multicast algorithms that deliver messages in a certain order must be devised. These algorithms are reviewed because they are able to satisfy different ordering properties that satisfy applications of different nature. In these algorithms, the delivery of a message may not immediately follow the receiving of a message to satisfy the specified ordering property.

In FIFO-ordered multicast, if two messages are multicast by the same process, then all processes will deliver those messages in the same order [3]. In the algorithm that ensures FIFO-ordering, every process p_i maintains the variables s_i and r_{ij}, where s_i represents the number of messages process p_i has sent to the group, and r_{ij} denotes the sequence number of the most recent message from process p_j that process p_i has delivered. For p_i to multicast a message m, it piggybacks s_i on the message as s_m. When p_j receives m, if $s_m = r_{ji} + 1$, then p_j immediately deliver the message because it is the subsequent one from p_i. Otherwise, it would queue the message and wait until $r_{ji} = s_m - 1$ to deliver it. This algorithm guarantees FIFO-ordering by the sequence number mechanism, and its reliability depends on the reliability of the underlying multicast. However, since FIFO-ordered multicast arbitrarily orders messages sent by different processes, it cannot satisfy the happen-before relation in the sequence of messages being delivered, which is problematic in certain applications.

To overcome the limitation of FIFO-ordered multicast, causally ordered multicast is devised such that message m_1 is delivered before message m_2 in all correct processes if the multicast of m_1 happens before the multicast of m_2 [3]. Essentially, the underlying algorithm uses vector timestamps to determine if a message is the subsequent one in the sequence of causal ordering. If so, the message is delivered immediately. Otherwise, the message is queued until it is the subsequent one. However, in casually ordered multicast, different processes may deliver messages in different orders. To understand why this is the case, consider a directed graph whose nodes are messages. A directed edge is formed between two messages with a causal dependency in between. A topological sorting of the messages must be a valid causal ordering of them. Since there may exist multiple ways to sort events topologically, processes may deliver messages in different order.

If the application demands that all processes must deliver messages in the exact same order, then it is necessary to use total order multicast. Unfortunately, devising an algorithm for total order multicast is non-trivial, and the problem is usually considered under the context of consensus.

3.4. Consensus

Many of the above problems involve algorithms that enable processes to agree on a certain value. The processes agree on which process to enter the critical section in mutual exclusion, which process to be the leader in elections, and the next message to deliver in total order multicast. As discussed above, these problems can be solved using a general consensus algorithm that enables processes to reach agreement on a certain value [1]. More specifically, after one or more processes propose a value, the consensus algorithm will decide on a value among the proposed values, and all correct processes will decide on the same value.

An algorithm that solves the consensus problem is chosen from the literature and reviewed because of its simplicity. This algorithm guarantees termination and correctness in the presence of f crash failures

exhibited by processes [3]. In essence, the algorithm proceeds in $f + 1$ rounds, and in each round, each process multicasts values that have not been sent in previous rounds. Every correct process will arrive at the same set of values even in the worst case, in which all f crashes occur during the rounds, because eventually the correct processes will be able to agree with the additional round. Then, the processes perform the majority function on the set of values, which returns the value that appears the most often in the set or some special value if no majority exists. Functions such as minimum, maximum, and median are applicable as well if the values can be sorted. As such, the correct processes arrive at the same value, and the consensus problem is solved. This algorithm relies on the assumption of synchrony because the notion of rounds is defined by physical time. This assumption is impractical in real systems because it does not align with the asynchronous behavior of networks. Therefore, it seems necessary to weaken the assumption of synchrony to one of asynchrony.

3.5. Paxos

Paxos is chosen for this review because it is the dominant offering in solving the distributed consensus problem in an asynchronous system, which has no assumptions about network latency and process execution speed [7, 8]. Paxos involves three roles, proposers, acceptors, and learners. A process can assume multiple roles at a time. If a process wants to be the proposer, perhaps due to speculating about the failure of the previous proposer from some timeout mechanism, the process multicasts a proposal consisting of a sequence number to all acceptors. The sequence number is unique to those other potential proposers can choose, which is ensured by proposers choosing their sequence numbers from disjoint sets. In addition, the sequence number must be greater than any sequence number the proposer has seen.

Upon receiving the proposal, if the sequence number in the proposal is higher than any sequence number the receiving acceptor has seen, then it sends a promise message to the proposer to accept the proposal, promising not to accept proposals consisting of lower sequence numbers. Otherwise, the acceptor sends a reject message to the proposer. If a promise message is to be sent, the acceptor attaches the latest value it has seen and the sequence number of the proposer who sent that value. This mechanism aims to ensure that current proposers can select the same value as the previous ones to continue the consensus established previously.

If the proposer receives promise messages from a majority of acceptors, it has been effectively elected. If any of the promise messages contain a value from previous rounds, the newly elected proposer must choose the value from the message containing the largest sequence number, which signifies the most recent previous proposer. Otherwise, any value can be chosen. After determining the value, the proposer multicasts the value to all other acceptors in an accept message. Once the proposer receives the acknowledgment from a majority of its acceptors, the proposer multicasts the value in a commit message to the learners, and consensus has been reached.

This algorithm ensures a single proposer in most cases, although it is possible for several to coexist due to the system model of asynchrony [8]. The algorithm guarantees correctness in the presence of multiple proposers, but progress is not necessarily ensured when two proposers keep issuing proposals with increasing sequence numbers. This is consistent with the FLP impossibility result obtained by Fisher, Lynch, and Paterson, stating that it is impossible for a deterministic algorithm to guarantee reaching consensus in an asynchronous system in the presence of failures [9]. Nevertheless, Paxos manages to terminate with high probability while guaranteeing correctness, which is usually sufficient for the purposes of most systems [3].

In practical systems, consensus algorithms are usually used to generate a sequence of values such as a replicated log that all processes agree on, which is analogous to total order multicast previously discussed. It is inefficient to run the entire Paxos algorithm repeatedly for every entry in the sequence, so

optimizations must be devised. Assuming that the proposer does not need to change between instances of Paxos, the election phase of the algorithm does not need to be repeated between instances [8], which significantly reduces the number of message exchanges and disk writes. This optimization is known as Multi-Paxos.

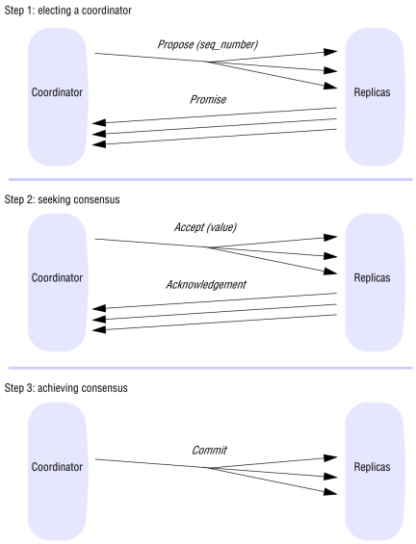

Fig. 4. Brief overview of the Paxos algorithm [3, 10].

Paxos is a complicated and subtle algorithm, and it is inevitably challenging for engineers to implement it in real systems. The team that developed Chubby, a distributed lock and file system that uses Paxos at its core, made adaptations to the original algorithm to suit the purposes of the system [10]. For example, because every cluster that uses the Paxos algorithm only involves 6 processes, the team made all processes other than the proposer, or the coordinator, both the acceptors and the learners. The correctness of this adaptation is ensured by an assumption made in the original algorithm that processes can assume multiple roles. Figure 4 provides a brief overview of the Paxos algorithm after this adaptation.

However, the team encountered numerous engineering challenges when implementing the algorithm. For example, disk corruption defeats the assumption of Paxos that processes have persistent memory. Another challenge they encountered was handling changes in group membership. This challenge could not be simply resolved, as Lamport suggested [8], by making group membership part of the state and changing it with ordinary state-machine commands. After altering the original Paxos algorithm to address the above challenges, the team arrived at an algorithm that is robust testing-wise but unproven mathematically, with somewhat satisfactory robustness.

3.6. Byzantine Generals Problem

All of the fault-tolerant algorithms above assume that processes fail by crashing, and in some cases, they would recover from failures and rejoin the system. However, in reality, it is possible for processes to fail in arbitrary ways, known as Byzantine faults [11]. For example, if the proposer in Paxos were to fail arbitrarily, it may send different values to different learners in the commit messages in the third phase, which undermines the algorithm's ability to reach consensus. Therefore, an algorithm that can reach consensus despite Byzantine faults must be devised to meet the requirements of certain applications.

The Byzantine Generals Problem is an analogy of a distributed system seeking consensus. In this analogy, a general sends a value to his $N-1$ lieutenants such that all loyal lieutenants agree on the same value, which is the value the general sent if he is loyal. If the commander is not loyal, the lieutenants must agree on some value, nevertheless.

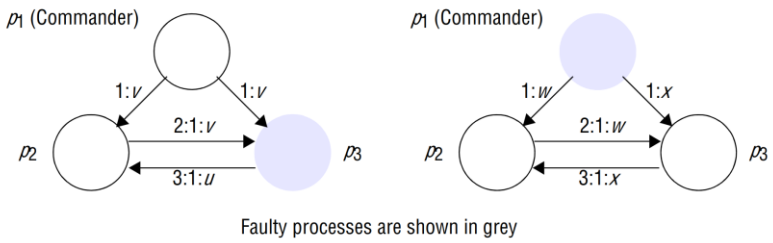

Fig. 5. Diagram for Byzantine Generals Problem [3].

In a situation involving three generals, one of whom is faulty, no solution is possible assuming the generals can only exchange oral messages, which do not allow the generals to prove that another general sent a certain message. Suppose p_1 is the commander, and p_2 and p_3 are his lieutenants. Consider the case in which p_3 is a traitor. p_1 sends the value v to p_2 and p_3. p_2 correctly echoes v to p_3, but p_3 sends some other value u to p_2. As a result, p_2 has received two conflicting values, u and v. Alternatively, consider the case in which the commander p_1 is the traitor. p_1 sends the value w to p_2 but some other value x to p_3. p_2 and p_3 correctly echo the value they have received from p_1 to each other. In this case, both p_2 and p_3 have received conflicting values, w and x. In the two cases, it is indistinguishable from the perspective of p_2 whether p_1 or p_3 is the traitor because it would receive conflicting values either way. Therefore, p_2 cannot choose a value among the two it has received. In fact, the Byzantine Generals Problem is unsolvable with $N = 3$ and $f = 1$, where f is the number of faulty generals. This intuitive observation can be generalized into a mathematical proof that states the Byzantine Generals Problem is unsolvable if $N \leq 3f$ but solvable if $N > 3f$ [12].

An algorithm was devised to solve the Byzantine Generals Problem when $N > 3f$ [11]. This algorithm is reviewed because it is the only algorithm that solves the Byzantine Generals Problem without cryptography. Processes recursively assume the role of commander and multicast the value they obtained from the majority function whose input is from previous rounds of recursions. Due to the recursive nature of the algorithm, it involves sending $O(N^{f+1})$ messages, which can be considered as multiplying an additional N for every round of recursion that aims to eliminate the impact of one traitor. Due to the expensive nature of the algorithm, more efficient solutions were devised using cryptography to achieve the assumption of signed messages, which allow processes to prove that a process sent a certain message.

Given an algorithm for the Byzantine Generals Problem, distributed consensus can be achieved in presence of arbitrary faults [11]. Every correct process maintains a vector v. After running the algorithm of the Byzantine Generals Problem, all correct processes set v_i to be the value the lieutenants agreed on with p_i being the commander. Due to the requirements of the Byzantine Generals Problem, v_i is precisely the value process p_i proposed if p_i is correct. Even if p_i is faulty, v_i is consistent on the vectors of all correct processes. As such, all correct processes obtain the same vector. They can perform the majority function on the vector to extract the most popular value, and consensus is achieved because all correct processes will reach the same value.

4. Conclusion

This is a review of some of the fundamental algorithms in distributed systems that aim to solve a variety of problems that include synchronization, ordering of events, mutual exclusion, election, ordered multicast, consensus, and Byzantine Generals Problem. In these algorithms, processes typically need to coordinate their actions to achieve some objective. Distributed systems engineers need to be familiar with these algorithms and be aware of their usage and limitations. Many other important algorithms exist for solving these problems in the literature, but the algorithms reviewed in this paper are to a large extent relevant to

the other algorithms.

Conflict of Interest

The author declares no conflict of interest.

References

[1] Kleppmann, Martin. Designing Data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. First edition, O'Reilly, 2017.

[2] Lamport, L. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558–65.

[3] Coulouris, George F., et al. *Distributed Systems: Concepts and Design*. Fifth edition, Addison-Wesley, 2012.

[4] Ricart, G, and Ashok K. A. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, vol. 24, no. 1, Jan. 1981, pp. 9–17.

[5] Maekawa, M. A √N Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, vol. 3, no. 2, May 1985, pp. 145–59.

[6] Birman, Kenneth P. The process group approach to reliable distributed computing. *Communications of the ACM*, vol. 36, no. 12, Dec. 1993, pp. 37–53.

[7] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems*, vol. 16, no. 2, May 1998, pp. 133–69.

[8] Lamport, L. "Paxos made simple." ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), Dec. 2001, pp. 51–58.

[9] Fischer, Michael J., et al. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, vol. 32, no. 2, Apr. 1985, pp. 374–82.

[10] Chandra, Tushar D., et al. "Paxos made live-An engineering perspective (2006 Invited Talk)." Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing, 2007.

[11] Lamport, L. et al. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, July 1982, pp. 382–401.

[12] Pease, M., et al. Reaching agreement in the presence of faults. *Journal of the ACM*, vol. 27, no. 2, Apr. 1980, pp. 228–34.