Impossibility of Distributed Consensus with One Faulty Process

Shashi Prakash

Abstract The consensus problem is a fundamental challenge in distributed computing where a group of processes must agree on a single value in an asynchronous system where some processes may be unreliable. This seminar thesis presents a proof that any protocol for solving the consensus problem is susceptible to non-termination, even with a single faulty process. In contrast, solutions exist for the related synchronous problem, known as the Byzantine Generals problem[1].

1 Introduction

The main fundamental problem in distributed consensus systems is reaching agreement among all the components in the system, despite the presence of failures and network delays. This is often referred to as the "consensus problem." The transaction commit problem is a classic example of the consensus problem in distributed systems. It refers to the process of ensuring that all copies of a distributed database eventually agree on whether or not a given transaction should be committed or rolled back.

A distributed database system is composed of multiple replicas, each maintaining a copy of the database. When a client initiates a transaction, the transaction is executed on one of the replicas, which is known as the coordinator. The coordinator then sends a commit request to all other replicas, known as the participants, asking them to commit the same transaction.

The participants can respond to the commit request in one of three ways: (1) they can vote "yes" to commit the transaction, (2) they can vote "no" to abort the transaction, or (3) they can vote "abstain" if they are unable to reach a decision. A consensus is reached when a majority of the replicas vote to commit or abort the transaction[2].We can understand with a simple example, three

Shashi Prakash
Paderborn University, e-mail: prakash@campus.uni-paderborn.de

nodes (A, B, and C) are involved in a transaction. Node A initiates the transaction and sends a request to nodes B and C to commit. Node B sends an acknowledgement to node A indicating that it is ready to commit. However, node C fails and is unable to send an acknowledgement. Node A is then left in a state of uncertainty as to whether the transaction should be committed or not. If node A decides to commit, it could lead to an inconsistent state, while if node A decides to abort, it could lead to unnecessary rollback of work. The consensus problem is to come up with an algorithm that ensures that all non-failing nodes will eventually agree on a common decision.

In a distributed system, failures can occur in various forms such as process failures, network splits, and lost, corrupted, or duplicated signals. One specific type of failure is known as Byzantine failure. This can be likened to a scenario where multiple divisions of an army are stationed outside of an enemy city, each led by a different commander. They must coordinate and agree on a common course of action, but some generals may betray their alliance and try to prevent a consensus from being reached[3].

Similarly, in a distributed system, there may be a Byzantine node or process that goes out of control and sends false signals and messages with malicious intent. To overcome this, an agreement protocol must be implemented that is reliable even in the presence of faulty processes. However, it's important to note that the protocol should be able to handle a specific number of faulty processes to prevent it from being overwhelmed.

The system model used in this seminar essay is a fully asynchronous system, where there are no assumptions about the relative speeds of processes or the delay time in delivering messages, and where processes do not have access to synchronized clocks, even a single unexpected process death can cause the system to fail. They also assume that the messaging system is reliable and that Byzantine failures are not taken into account. This means that even if all other conditions are met, a distributed commit mechanism may still fail to establish consensus if a single process is stopped at the crucial time. The authors also make it clear that without additional computing environment assumptions or stricter limitations on the types of failures that may be allowed, this problem cannot be effectively solved. This is because it is impossible for one process to identify whether another has died or is just operating extremely slowly, as there is no way to detect the death of a process[1].

This seminar essay provides the five following sections. Second, give an overview of the related work. System model and consensus are explained in the third section and main result are described in the forth section. Fifth section is explained the initial dead process and in section sixth, I conclude my seminar essay.

2 Related Work and Impact of FLP Result on the Distributed System

In a distributed system, an asynchronous system is a system in which the processes do not need to operate in a strictly synchronised manner. This means that the processes can execute independently of each other, and do not need to coordinate their actions through the use of clocks or other synchronisation mechanisms.

On the other hand, a synchronous system is a system in which the processes do need to operate in a strictly synchronised manner. This means that the processes must coordinate their actions using clocks or other synchronisation mechanisms, and may need to wait for certain events to occur before they can take certain actions.

There are trade-offs to both asynchronous and synchronous systems. Asynchronous systems tend to be more resilient to failures, as they do not rely on strict synchronisation and can continue to operate even if some processes fail or become slow. However, asynchronous systems can be more difficult to design and debug, as the lack of synchronisation can make it harder to reason about the behaviour of the system.

Synchronous systems, on the other hand, tend to be easier to design and debug, as the use of synchronisation mechanisms can make it easier to reason about the behaviour of the system. However, synchronous systems are more vulnerable to failures, as the reliance on synchronisation means that the system can become stuck or halt if certain processes fail or become slow[9].

The **FLP** (Fischer, Lynch, and Paterson) result, published in a paper "Impossibility of Distributed Consensus with One Faulty Process" by Fischer, Lynch, and Paterson in 1985, states that in an asynchronous distributed system, it is impossible for all non-faulty processes to agree on a common value if even one process may fail.

The FLP result has a significant impact on the design and implementation of distributed systems. It implies that it is impossible to design a consensus protocol that guarantees that all non-faulty processes will agree on a common value, even if only one process may fail. This means that distributed systems must be designed to tolerate failures and to handle the possibility of different processes having different values.

The distributed consensus problem has been extensively studied in the literature and various solutions have been proposed.

One of the early solutions to the problem is the Paxos, a consensus algorithm proposed by Leslie Lamport in 1989, is a solution to the problem of achieving consensus in a network of unreliable processors. The algorithm uses a leader-based approach, where a leader is responsible for choosing the consensus value. This ensures that the consensus value is chosen only by a correct leader. Paxos ensures that all nodes in a distributed system agree on the same value, even in the presence of failures or network delays. The algorithm is composed of three phases: Proposal, Preparation, and Acceptance. Once a majority of acceptors have accepted the proposal, it becomes the consensus value[4]. The algorithm has been widely studied and has been used in various practical systems such as Google's Chubby lock service and the Zookeeper coordination service.

Another well-known solution is the Raft algorithm, proposed by Diego Ongaro and John Ousterhout in 2014. Raft is a consensus algorithm that is designed to be easy to understand and implement. It uses a leader-based approach and provides a set of safety and liveness guarantees[6]. The algorithm has been widely used in various practical systems such as etcd and Consul.

In recent years, there has been a renewed interest in the problem of distributed consensus and various new solutions have been proposed. One of these is the BFT-SMaRt algorithm, proposed by Miguel Castro and Barbara Liskov in 2002. BFT-SMaRt is a Byzantine fault-tolerant consensus algorithm that is designed to work in a partially synchronous network[7]. The algorithm has been widely studied and has been used in various practical systems such as the Hyperledger Fabric blockchain platform.

Another recent solution is the HotStuff algorithm, proposed by Yin and Dahlia Malkhi in 2018. HotStuff is a Byzantine fault-tolerant consensus algorithm that is designed to work in a partially synchronous network. It is based on the concept of "hot" leaders and provides a set of safety and liveness guarantees[8]. The algorithm has been widely used in various practical systems such as the Dfinity blockchain platform.

The FLP result had a significant impact on the field of distributed computing. The study, which established the impossibility of achieving consensus in an asynchronous distributed system with even one faulty process, was recognized as having the greatest impact on the field and was awarded the Dijkstra Award in 2001. The authors of the paper, Michael Fischer, Nancy Lynch, and Mike Paterson, are still considered prominent scholars in the field of distributed algorithms, with Nancy Lynch particularly known for her significant contributions to the development of distributed algorithms.

3 System Model and Consensus

3.1 System model

In the original paper, the authors propose a system model for distributed consensus algorithms. The system model is a set of assumptions about the underlying distributed system that the algorithm is intended to run on.

The model consists of a network of dispersed processors that act as vertices in a connected graph, with communication links acting as the edges along which messages can be conveyed. The authors consider an asynchronous model, which is a family of models with specific timing characteristics. In an asynchronous model, there is no upper limit on how long it may take for processors to receive, process, and react to an incoming message. As a result, it is impossible to determine whether a processor has malfunctioned or is merely taking a long time to process information.

The authors of the consensus algorithm under consideration make the assumption that the communication channels between the processors are reliable. This implies that the transmission of messages between the processors is error-free, and that there are no omissions or duplications of messages. Additionally, the authors take into account the fail-stop model, which posits that it is acceptable for processors to malfunction and subsequently cease to function properly. This means that a malfunctioning processor will no longer be able to participate in the consensus process and its failure will not impact the ability of the remaining processors to reach consensus. *Reliability* in this context is crucial for the correct functioning of the consensus algorithm, as it ensures that all messages are correctly transmitted and received, and that the failure of any processor does not disrupt the consensus process[1].

The authors use this system model to prove that it is impossible to achieve distributed consensus with one faulty process. The authors demonstrate that there are some initial configurations where the decision is not yet determined and they design an admissible run that never takes a step that would force the system to make a certain decision. This highlights the limitations of consensus protocols and the importance of robustness and fault tolerance in the design and implementation of such systems.

In a system model, processes are modeled as automata, which are machines with a finite or infinite number of states. These processes communicate with each other by sending and receiving messages in an atomic step. An atomic step refers to a single unit of execution in which a process can attempt to receive a message, perform local computation based on the message received, and send a finite number of messages to other processes. The system is assumed to have an atomic broadcast capability, meaning that a process can send the same message to all other processes in one step, and if any non-faulty process receives the message, all non-faulty processes will receive it as well. However, it is also assumed that messages may be delayed, delivered out of order, and may take an arbitrarily long time to be delivered, but will eventually be delivered as long as the destination process continues to make attempts to receive them[1].

3.2 Consensus

Consensus is a critical aspect of distributed systems, where multiple computers in different locations work together to agree on a common value or decision. This is typically achieved through a voting process, where each computer "votes" for a particular value. The consensus algorithm then selects the final value based on the majority of votes. In simpler terms, consensus is the process of ensuring that all the computers in a network come to an agreement on a single value. If no value is proposed, then no value will be chosen. Once a value is chosen, all the computers should be able to learn and agree on the chosen value[5].

There are many practical applications of consensus in distributed systems. For example, in a distributed database system, a consensus algorithm might be used to determine whether or not to commit a transaction. This ensures that all copies of the database are in agreement before the transaction is committed, avoiding inconsistencies and conflicts.

In addition, consensus algorithms are also used in distributed systems to ensure that all processors have the same view of shared resources, such as memory or network links. This is particularly important in systems where there are multiple nodes that can make updates to shared resources simultaneously, as it ensures that all updates are properly coordinated and that all nodes have the same view of the state of the system.

Overall, the ability to reach a consensus is a critical aspect of distributed systems, as it enables coordination and cooperation among multiple processors and ensures that all nodes have the same view of the system. It is a fundamental building block that is used in many distributed systems and applications, making it an important area of research.

3.3 Consensus Protocols

A consensus protocol **P** is a distributed system of **N** processes, where **N** is at least 2. These processes operate independently of each other and do not have a fixed schedule or pattern of execution, which is referred to as being *asynchronous*. Each process, designated as p, has a one-bit *input register*, designated as x_p and an *output register*, designated as y_p that can hold values of "b", "0", or "1". The input and output registers, along with the program counter and internal storage, make up the *internal state* of each process. The *initial states* of the consensus protocol **P** prescribe fixed starting values for all registers except for the input register. The output register begins with the value b, indicating that its value is not yet determined and could be either 0 or 1. The processes in the protocol operate in a deterministic manner, following a *transition function*. This function determines the behavior of each process based on the current internal state and input it receives. Once a process reaches a *decision state*, where the output register has a value of 0 or 1, the transition function can no longer change the output register's value, as it is "write-once." The entire system **P**, is defined by the transition functions associated with each process and the initial values of the input registers.

The processes within the protocol communicate with one another by sending messages. Each message consists of two elements: (p), representing the *destination process*, and (m), representing the message's content or *message value*, which is chosen from a fixed universe M. The universe of possible message values, M, is fixed, meaning that the set of possible values for the message content (m) is limited to a specific set and does not change. The message system also maintains a record of all messages that have been sent but not yet delivered, referred to as the *message buffer*.The message buffer supports two abstract operations:

send(p,m): This operation inserts the message (p,m) into the message buffer.

receive(p): This operation either returns the special null marker $\emptyset$,[1] indicating that there are no messages in the buffer directed towards p, or it returns one of the messages for processor P at random (and removes it from the message buffer).

Thus,the message system is a non-deterministic system, meaning that the order of message delivery is not predetermined and may vary depending on the system's state. The system is subject to the condition that if the receive(p) function is called an infinite number of times, every message(p, m) in the message buffer will eventually be delivered. However, the system is not required to deliver messages immediately and may return $\emptyset$, indicating that no message is available, a finite number of times before delivering a message. In other words, the message system may choose to delay the delivery of certain messages, and it is not guaranteed that all messages will be delivered in the order they were sent[1].

In the consensus protocol, a *configuration* C represents the internal state of each processor, their memory, and the step of the algorithm they are on, including information in the message buffer. An initial configuration is when the message buffer is empty and all processes are in their initial state. The system progresses through configurations by a processor p performing receive(p) and transitioning to another internal state. Configurations change only by processors receiving a message or null from the message buffer, each step is uniquely defined by the message received and the processor that received it. The pair (p,m) is referred to as an *event* in the paper, configurations move from one to another through events, and the notation e(C) denotes that event e = (p,m) can be applied to configuration C. It's worth noting that the process can always proceed to the next step as the event (p, $\emptyset$) can always be applied to C.

In the consensus protocol, a *schedule* is a sequence, either finite or infinite, of events that can be applied to a configuration C in a specific order. This sequence of events is referred to as a *run* and the resulting configuration, denoted as σ(C), is considered reachable from C if the schedule, σ, is a finite sequence. Any configuration that can be reached from an initial configuration is referred to as being *accessible*. In the rest of this section, it is assumed that all configurations mentioned are accessible.

The following lemma illustrates the *commutativity* property of schedules[1]. This property states that the order in which events are applied to a configuration does not affect the final outcome, meaning that the resulting configuration is the same regardless of the order in which events are applied. In other words, if two schedules, σ and σ', are applied to the same configuration C, the resulting configurations, $\sigma(C)$ and $\sigma'(C)$, should be the same. This property is important in the consensus protocol to get the main result.

Lemma 1. *Suppose that from some configuration C, the schedules σ_1, σ_2 lead to configurations C_1, C_2, respectively. If the sets of processes taking steps in σ_1 and σ_2, respectively, are disjoint, then σ_2 can be applied to C_1 and σ_1 can be applied to C_2,and both lead to the same configuration C_3 [1].*

Proof. We start from a configuration C and apply two schedules of events, σ_1 and σ_2, to the configuration, leading to configurations C_1 and C_2 respectively (as shown in Figure 1). Additionally, all processes take a step in σ_1 and σ_2 and the processes that take steps in σ_1 and σ_2 are mutually disjoint, meaning that they do not overlap.

We begin by defining a function P(σ) that defines all the processes that take steps in schedule σ. Then, P(σ_1) and P(σ_2) represent the sets of processes that take steps in schedules σ_1 and σ_2 respectively. The intersection of these sets, P(σ_1) $\cap$ P(σ_2), is an empty set, denoted as $\emptyset$. This means that schedule σ_1 works on one set of processes, and schedule σ_2 works on a different set of processes, and these two sets are disjoint. □

A process p is considered *nonfaulty* in a run if it continues to take steps indefinitely. In contrast, if a process p fails to take infinitely many steps, it is considered faulty in that run. This means that a nonfaulty process p is assumed to operate normally and follows the algorithm as expected, while a *faulty* process p deviates from the algorithm and may exhibit unpredictable behavior. An *admissible run* is one in which every message is finally delivered and there is no more than one process that is faulty, in accordance with the failure constraints of the system model.

A run is considered a *deciding run* when one of the processes in the system reaches a state where it has made a decision. Conversely, we say that a consensus protocol is completely correct when each admissible run is a deciding run.A deciding run refers to a scenario in which all non-faulty processes in the system have completed their steps and reached a consensus on a certain value. On the other hand, an admissible run refers to a scenario in which all messages are finally delivered and there is no more than one process that is faulty, as defined by the failure constraints of the system model.

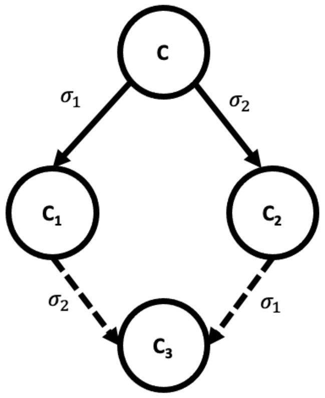

Fig. 1

4 Main Result

The goal of this theorem is to demonstrate that there exists an admissible run that is not a deciding run. This means that even if the system satisfies the failure constraints and all messages are delivered, there still may be a run where the processes do not reach a decision.

This theorem highlights the limitations of consensus algorithms and the importance of understanding the potential for non-deciding runs in the design and implementation of such protocols. It also emphasizes the need for robustness and fault tolerance in distributed systems, as the presence of faulty processes can affect the decision-making process.

Theorem 1. *No consensus protocol is totally correct in spite of one fault.[1]*

The statement of "Theorem 1" means that it is not possible for any consensus protocol to guarantee that a decision will be reached in all scenarios, even when there is only one faulty process present.

Proof. The main objective of this proof is to show that there are situations in which the protocol is permanently indecisive. To achieve this, the authors make two main claims. Firstly, they assert that there exist some initial configurations where the decision is not yet determined. Secondly, they design an admissible run that never takes a step that would force the system to make a certain decision.

To formalize these claims, the authors defined the set V which represents the possible decision values of configurations that can be reached from configuration C. If the size of the set is 1, then the configuration C is considered univalent. If the size of the set is 0, then the configuration C is considered bad, indicating that the algorithm did not arrive at any agreement. □

Lemma 2. *P has a bivalent initial configuration. [1]*

Proof. The second lemma's objective is to demonstrate that there are certain initial configurations, where the decision is not predetermined but rather determined by the sequence of events and failures. For example, the authors demonstrate that in an initial configuration consisting of two processors with starting values of 0 and 1, respectively (See in Figure 2), the outcome is dependent not only on the initial settings of the processors but also on the sequence in which messages are delivered and if any of the processors fail. This is due to the built-in nondeterminism in the asynchronous system model.

To demonstrate this, the authors are assuming that for every initial configuration of values in the processors, there is a predetermined sequence of execution or set of operations that will occur. This means that given a specific initial configuration, there is a specific outcome that will be produced.

They also argue that by the principle of validity, all possible outcomes (in this case, either "0" or "1") must be able to be achieved. Therefore, there must be some initial configurations that result in 0 being decided and others that result in 1 being decided.

To understand the relationship between different configurations and their associated outcomes, the authors suggest ordering the configurations in a chain.(See in Figure 3) Two configurations are considered to be next to each other if they only differ by one value. This means that the only difference between two adjacent configurations is the initial value of one processor.(See in Figure 4) This allows for a clear understanding of how small changes in the initial values can lead to different outcomes.

At some point in this chain, an initial configuration that decides 0 must be next to an initial configuration that decides 1. This means that they must be different by only one initial value. The authors call the configuration that decides "0" C_0 and the configuration that decides "1" C_1. They also call the processor whose starting value is unique to each configuration p. Starting with C_0, there must be a run that chooses 0, even if p fails initially. This is because we allow one of the processors to fail at a time.

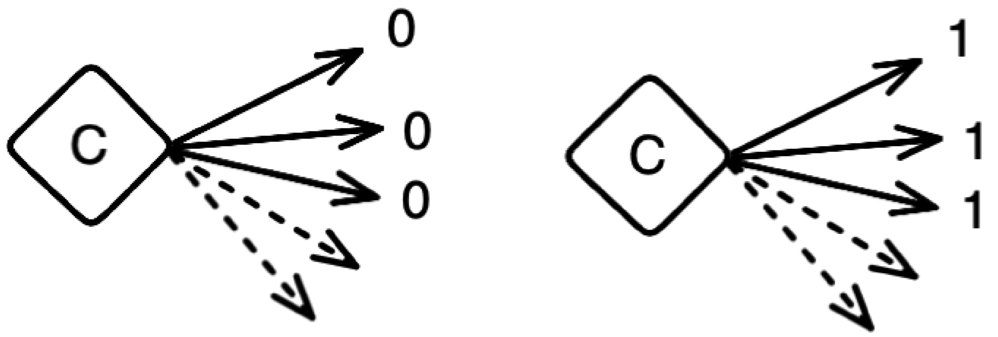

Fig. 2 Initial configurations 0 and 1

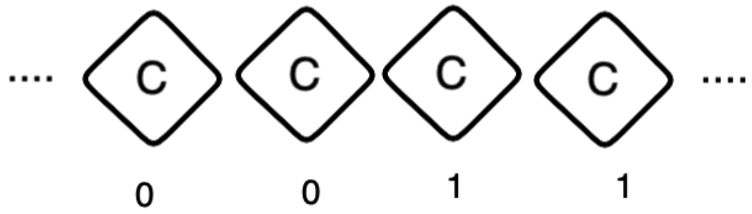

Fig. 3 N processes in a chain form

The key takeaway is that the authors consider two initial configurations C_0 and C_1, that differ only in the initial value of a single process p. They argue that if there is an admissible deciding run from C_0 in which process p takes no steps, then the same schedule can be applied to C_1 and the resulting configurations will be identical except for the internal state of process p. Since both runs eventually reach the same decision value.(See in Figure 5) It contradicts the assumption that the outcome of the consensus algorithm is solely controlled by the starting configuration. □

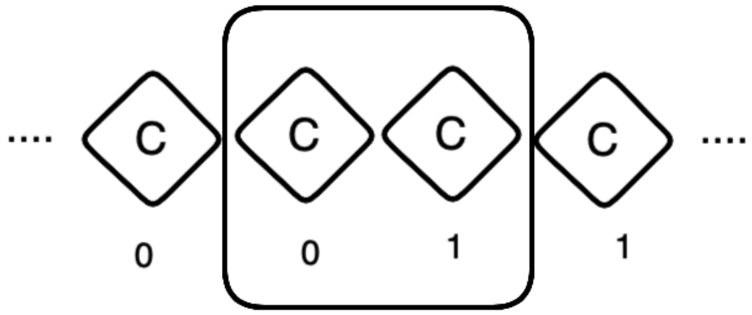

Fig. 4 Adjacent C differ only by one process

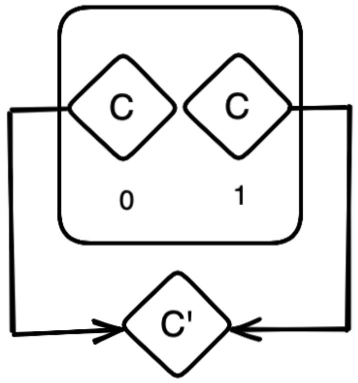

Fig. 5 Only p is different in the state but p is crashed

Lemma 3. *Let C be a bivalent configuration of P, and let e = (p, m) be an event that is applicable to C. Let* ***C*** *be the set of configurations reachable from C without applying e, and let* ***D*** *= e(****C****) = e(E) | E ∈* ***C*** *and e is applicable to E. Then,* ***D*** *contains a bivalent configuration[1].*

Proof. We assume that **D** does not have any bivalent configurations. A contradiction results from this, as the proof shows. If an e = (p,m) is applicable to a starting configuration C and messages can be delayed arbitrarily, then that rule should be applicable to every configuration in a set **C**. However, the assumption is made that another set **D** contains only univalent configurations (configurations that are either 0-valent or 1-valent, but not both).

The argument then proceeds to show that this assumption is false by considering a specific configuration E_i that is reachable from the starting configuration C, i=0,1[1]. If E_i is already in the set **D**, then applying the event e to E_i results in another configuration F_i that is also in the set **D**. If E_i is not in **C**, then it must be reachable from $F_i \in$ **D**. In either case, F_i must also be i-valent (0-valent or 1-valent) because it is not bivalent and is reachable from E_i. Since F_i is in the set **D**, this means that **D** contains both 0-valent and 1-valent configurations, which contradicts the assumption that **D** contains only univalent configurations. Therefore, the assumption that **D** contains only univalent configurations must be false. □

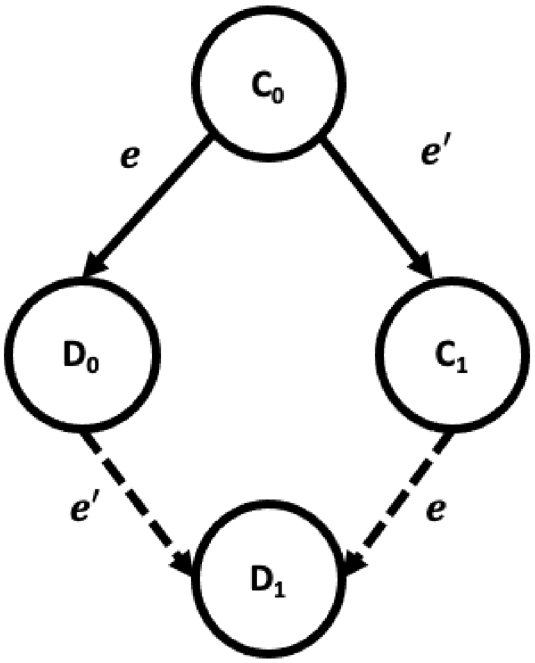

Fig. 6

Due to the fact that just one message delivery separates C_0 and C_1, they are referred to as "neighbors." Consider the message e′=e′(p′,m′) that separates them now. Take into account the two potential outcomes for e′s final destination p′.

Case 1: If **p' $\neq$ p**[1], In a distributed system, there may be situations where two configurations, D_0 and D_1 are separated by a message that is intended for a different processor. If this is the case, we have the ability to reach D_1 from D_0 by receiving the message e′=e′(p′,m′) at D_0. However, this is only possible if the destination of the message is different than e. Because all processors see the same sequence of events, we can take the messages in any order and still reach the same configuration. This is known as the commutativity property of schedules. However, if D_0 is 0-valent and D_1 is 1-valent, then it is not possible to reach D_1 from D_0 as it would suggest that the protocol has changed direction. This creates a contradiction, as shown in Figure 6.

Case 2 : if **p' = p**[1], By considering a finite deciding run starting from a configuration C_0, where getting a specific message e results in a 0-valent configuration and a process p does not take any steps in this run. We can call the *resulting configuration A*. We can also apply this run to either D_0 or D_1 to generate two new configurations E_0 and E_1. Since schedules are commutative, we can also get e at A to get E_0 and e$'$ followed by e at A to get E_1. This means that A is bivalent, which contradicts the fact that the run to A is the one that decides. Thus, we have shown that there must be a bivalent configuration in D through the use of contradiction. (See Figure 7)[1].

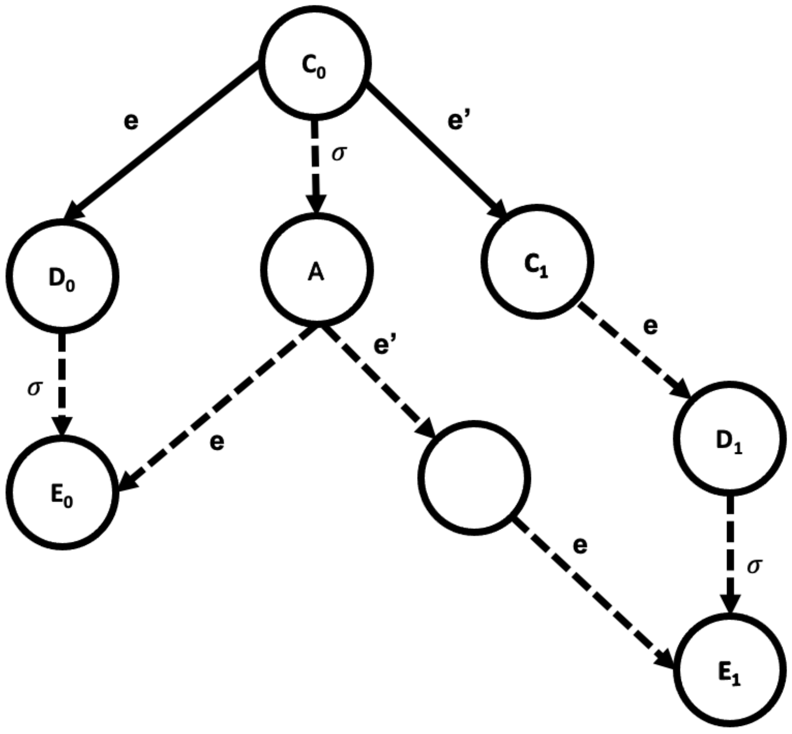

Fig. 7

The last step in the proof is to show that it is possible to create an infinite number of non-deciding runs from any deciding run. To achieve this, the runs must be constructed in such a way that a decision is never made, which would require entering a univalent configuration.

Starting with the initial configuration C_0, which has been proven to be bivalent (per the Lemma 2), the processors in the system can be arranged in a queue and instructed to process messages in the order they were received. Once the processors have finished processing their messages, they are moved to the end of the line. This ensures that every message will eventually be delivered.

The protocol in question is being analyzed using a method called the Lemma 3. This method involves considering the earliest message in a queue to be processed by the first processor, and using this information to reach a bivalent configuration C_1 that is accessible from the initial configuration C_0. By continuing to use this method, it is possible to reach additional bivalent configurations (C_2,

C_3, etc.). However, it is also possible for this process to continue indefinitely without reaching a final conclusion. Despite this, the protocol is deemed to be totally correct. □

5 Initially dead processes

The protocol presented in this section is designed to solve the consensus issue among N processes, where a majority of the processes are non-faulty and no process dies during the execution of the protocol. The protocol is divided into two phases. In the first phase, the processes create a directed graph G, where each process is represented by a node and an edge is formed when one process sends a message to another. Each process then broadcasts its process number and listens for L-1 additional processes, where L = [(N + 1)/2][1].

In the second phase, the processes create G+, the transitive closure of G, which means that each process is aware of all edges (j, k) that are incident on k in G+, as well as the starting values of all such j. This allows each process to know all of its ancestors and the edges that connect to them. At the end of this phase, each process is able to make a decision based on the initial values of the other processes in the initial clique, following any predetermined rule.

In the second stage, each process broadcasts its process number, starting value, and the names of the L-1 processes it heard from in the previous stage to all other processes. It then waits until it receives stage 2 messages from all of its descendants in G. Initially, it only knows of the L-1 processes it heard directly during the first stage, but as it receives more messages, it learns about new ancestors. The process continues to wait until it has received updates or messages from all known processes.

At this stage, each process becomes aware of all edges (j, k) that are incident on k in G+ and the starting values of all such j. The process then makes a decision based on the initial values of the other processes in the initial clique, and since all processes are aware of the starting values of all initial clique members, they all reach the same conclusion. The protocol is guaranteed to be accurate as it is based on the theorem that a node k is in an initial clique if it is an ancestor of every node j that is an ancestor of k. The clique has a cardinality of at least L, and every process that completes the second stage is aware of the precise set of processes that compose it.[1].

After the second stage, each process makes a decision based on the initial values of the other processes in the initial clique, using a predetermined rule. All processes have knowledge of the initial values of all members in the initial clique, allowing them to reach a consensus. The protocol's ability to achieve consensus is demonstrated by the theorem.

Theorem 2. *There is a consensus procedure that is only partly right. In this protocol, all non-faulty processes will always arrive at a decision, given that no processes will die while it is being executed and that an absolute majority of the processes will be alive when it begins[1].*

6 Conclusion

The FLP impossibility result relies on two key lemmas to demonstrate that consensus is impossible in an asynchronous system. The first lemma establishes that there exist initial configurations of the system where the outcome of consensus is not predetermined, and that the possibility of failures is crucial to this lemma. This is because if there were no failures, the current state of the network would be sufficient to reach a decision, as message delays could be recognized for what they are.

The second lemma demonstrates that it is always possible to achieve another bivalent configuration by delaying a pending message if one starts in a bivalent configuration, even if one does not start in a bivalent configuration. This can continue indefinitely, meaning that no protocol can ensure that a univalent state will ever be reached.

In practical terms, this means that while the risk of entering an endless loop of undecision may be made arbitrarily small, it is still a possibility. Additionally, in real-world systems, asynchronous behavior is not always guaranteed. Understanding these limitations is important for identifying and addressing issues that may arise during the consensus protocol, especially when the system is livelocking and the cause is unclear.

References

1. Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." Journal of the ACM (JACM) 32.2 (1985): 374-382.
2. C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction management in the R* distributed database management system. ACM Trans. Database Syst. 11, 4 (Dec. 1986), 378–396. https://doi.org/10.1145/7239.7266
3. Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." Concurrency: the works of leslie lamport. (2019). 203-226.
4. Lamport, Leslie. "The part-time parliament." Concurrency: the Works of Leslie Lamport. (2019). 277-317.
5. Lamport, Leslie. "Paxos made simple." ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) (2001): 51-58.
6. Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." 2014 USENIX Annual Technical Conference (USENIXATC 14). (2014).
7. Castro, Miguel, and Barbara Liskov. "Practical byzantine fault tolerance." OsDI. Vol. 99. No. 1999. (1999).
8. Yin, Maofan, et al. "HotStuff: BFT consensus in the lens of blockchain." arXiv preprint arXiv:1803.05069 (2018).
9. Cristian, Flaviu. "Synchronous and asynchronous." Communications of the ACM 39.4 (1996): 88-97.