# Let's Agree to Agree: Consensus in a Faulty System

Colin James Luha Yang Wesley Yang
Vanessa Heynes Arisha Khan Hanane Akeel

2025

Integrative Exercise Advised by Professor Tanya Amert

Dept. of Computer Science
Carleton College

# 1 What is Consensus?

The consensus problem is an integral part of distributed system computing; a style of computing where tasks and/or processes are divided across multiple independent computers. These processes are also referred to as nodes. The consensus problem is tasked with ensuring that these independent computers agree upon certain values and processes that the computers are performing. This agreement could be on a wide variety of things, ranging from eccurencies' blockchain receipts to changing a social media password on regional servers to banks agreeing on the money taken out of an account. For now, let us focus on the example of taking money out of a bank account.

There a several parts of this transaction. First you, the client, takes money out of your bank account. After, some ammount of computation occurs on the server you are accessing in order to take the money out. Now, this server must communicate with every other server that holds your bank account information. How do you get such servers to agree? The answer is consensus algorithms!

Consensus algorithms are defined as a mechanism that enables multiple nodes to agree on a single state or value, ensuring consistency and reliability even in the presence of failures or malicious actors. They are tasked with coordinating every process of a distributed system to communicate with each other, vote upon the correct value, and update all nodes to carry the same value.

This process is incredibly important, allowing almost every distributed system to operate correctly. So the question is asked: What makes this process so complicated?

Let us consider two types of failures that processes could experience. One type could be that the process dies or experiences network failure, in which it would lose its ability to communicate with other processes. This type of fault is called network failure or fail-stop. With the other fault being malicious, in which it communicates that it is correct, but transmits incorrect information. This type of fault is called malicious nodes or lying nodes. These faults along with several other challenges makes consensus algorithms necessary in distributed computing as well as complicated to implement.

# 2 Type of Consensus Algorithms

## 2.1 Leaderless and Leader-Based

Consensus algorithms can be broadly categorized into leader-based and leaderless approaches, each with its own advantages and trade-offs in ensuring agreement among distributed systems. Leader-based algorithms use a single node as a leader. This node will propose values and coordinate agreement inde-

pendently from the rest of the nodes in the system. Additionally, leader-based algorithms typically implement leader election or have a system of transitioning leaders. Leader-based algorithms are typically more efficient, however there is a possibility of a single point of failure. This makes leader-based algorithms more vulnerable to attack and bottle necking.

Another category of consensus algorithms are leaderless protocols. Leaderless algorithms do not have a single leader, instead distributing decision making across all nodes in the system. These types of algorithms increase fault tolerance, however usually increase latency due to higher amounts of work needing to be done.

In our project, we implemented both types, in an attempt to learn about the benefits and drawbacks of both. After implementing one of both, we decided as a group that leader-based algorithms are better suited for our purposes, offering faster speed, fault tolerance, and simpler implementation.

### 2.2 Synchronous and Asynchronous

Consensus algorithms operate under different timing assumptions, categorized as synchronous or asynchronous. In synchronous systems, there is a known upper bound on message transmission and processing times. Nodes execute steps in lockstep, ensuring predictable communication and agreement. This model simplifies consensus but is unrealistic in many real-world networks, where delays can vary.

In contrast, asynchronous systems assume no fixed time limits for communication or execution. Nodes may operate at different speeds, messages can be delayed indefinitely, and failures are harder to detect. Algorithms in this model, such as those used in blockchain networks, must tolerate uncertainty and often rely on probabilistic guarantees rather than strict agreement within a bounded time frame.

In our project, we chose to implement asynchronous consensus algorithms as they are more commonly implemented and model real world distributed systems more accurately.

## 3 Implemented Algorithms

### 3.1 Leaderless Byzantine Paxos

Leaderless Byzantine Paxos is a consensus protocol designed to operate without a single leader while tolerating Byzantine faults, where nodes may behave maliciously. The protocol relies on a quorum-based approach where multiple nodes independently propose values and validate others' proposals to

achieve agreement. A quorum-based approach is defined as needing a calculated number of correct nodes in order to achieve consensus. Each node collects votes from a majority, ensuring consistency despite faulty participants. This structure enhances fault tolerance and removes leader bottlenecks, making it ideal for decentralized environments. However, the absence of a leader increases coordination overhead, leading to higher communication complexity and slower agreement times compared to leader-based alternatives.

### 3.2 RAFT

Raft is a leader-based consensus algorithm designed for easier understanding and implementation compared to Paxos. It operates by electing a leader that manages log replication across nodes, ensuring consistency. The algorithm consists of three main stages: leader election, where nodes vote to select a leader; log replication, where the leader receives and distributes client requests to followers; and commitment, where entries are considered final once a majority confirms them. Raft enhances fault tolerance by allowing seamless leader reelection in case of failure. Its structured approach simplifies consensus implementation but introduces a single point of failure, as leader crashes can briefly disrupt system performance. Additionally, raft consensus algorithm only accounts for network failure nodes, and not malicious nodes.

### 3.3 HotStuff

HotStuff is a leader-based Byzantine Fault Tolerant (BFT) consensus algorithm designed for high scalability and efficiency in distributed systems. It operates in a series of rounds where a leader proposes a block, and nodes vote in a four-phase process. Unlike the previous consensus algorithms, HotStuff reduces communication complexity to linear levels, making it well-suited for blockchain and large-scale applications. Its modular design enables dynamic leader rotation, improving resilience against failures.

HotStuff is implemented using a chained structure, where each proposal extends the previous one, creating a continuous flow of blocks. This approach significantly reduces latency compared to Raft or leaderless byzantine paxos. Instead of requiring multiple rounds of communication per decision, HotStuff allows nodes to process new proposals while finalizing previous ones, ensuring a steady and efficient consensus flow.

The protocol follows a four-phase commit model in which nodes prepare, precommit, commit, and finally have a leader decide on proposed blocks. In the Prepare Phase, the leader proposes a block, and nodes validate it. During the Pre-Commit Phase, nodes exchange votes to confirm that a majority supports the proposal. Then, in the Commit Phase, the decision is finalized. Finally, in the decide phase, the leader node broadcasts the agreed upon message to all nodes, updating their information. This streamlined design makes it highly

scalable and suitable for blockchain applications requiring high throughput and security. However, its reliance on a single leader per round can introduce bottlenecks, and leader handovers may cause temporary slowdowns under adverse network conditions. Despite these challenges, HotStuff's ability to maintain strong consistency and high efficiency made it a compelling choice for us to choose over other algorithms.

# 4 The Fail-Stop Case

## 4.1 Fail-Stop

Consider an asynchronous model of $n$ fully interconnected processes. These processes will individually be called nodes. These nodes have the ability to communicate with each other with messages that contain values. The goal of these nodes is to have all processes agree on a single value by reaching a certain quorum of votes. However, some nodes can be fail-stop, where they experience network failure or die, preventing them from sending messages. In this section, we aim to identify the upper bound for how many fail-stop nodes can occur while a protocol can still continue to function, reaching consensus on a correct value.

## 4.2 Definitions and Notations

In order to prove the upper bound for $k$ faulty nodes for consensus algorithms in the fail stop case, let us define what type of distributed systems we will use. Consider $n$ interconnected processes to have 2 distinguishable memory locations input $i_p$ and decision $d_p$ with $d_p$ only being assigned a value in the set 0,1. Additionally once $i_p$ and $d_p$ have been decided, it cannot be changed by any process including during the communication process between nodes.

Another important definition will be a $\sigma$ schedule which will represent a series of atomic steps that lead from a C configuration to a D configuration such that $C \overset{\sigma}{\vdash} D$ where C is a starting configuration and D is the configuration that schedule $\sigma$ leads C to become. Atomic steps are defined as a process trying to receive a message, perform an arbitrary computation, and sending a finite set of messages. We will also assume the all consensus protocols and distributed systems will follow fair scheduling as defined in Section 3.3.

## 4.3 Fair schedule

Let us fully define the schedules and types of schedules that we will be using. In these proofs, we will be assuming fair schedules. This means that they meet several conditions as defined here.

First, the protocol for a consensus algorithm will operate in rounds such that in round $t$ a process will send a message to every other process, waiting for $n-k$ messages before reaching D. Only then will the process move to round $t+1$. This state in round $t+1$ is a function of the old state and the messages in round $t$. After proving the fail-stop case, we will be able to claim that at each state, they will not wait for more than $n-k$ node messages due to the possibility that they will never receive more than n-k messages.

Now we will define $R(q,p,t)$ as the event in which p receives a message from q in round t. It can be said that a scheduler is fair if it meets the following conditions: 1.) For any processes p,q at round t, there is a positive constant $\epsilon$ such that $Pr[R(q,p,t)] > t$. 2.) For any distinct processes $r,q,p$ at round $t$, the event $R(q,p,t)$ and R(r,p,t) are distinct. These conditions ensure that for any round $t$ there is a constant probability that all processes receives $n-k$ messages from the same set of correct processes.

## 4.4 K-Resilient Consensus Protocol

To begin proving the qourum sizes of different fault types, let us first look at how to define correctness. Instead of correctness we will define k-resilient consensus protocols. Where $k$ is the maximum number of faulty nodes possible while the consensus protocol will still carry out correctly. According to Toueg et al. for a protocol to be k-resilient it must satisfy three properties at a given $k$. These properties include:

- 1.) Bivalence: If all processes are correct, both $F^1$ and $F^0$ where 0 and 1 are the decisions for configurations are reachable. F is defined as reachable if there is some initial configuration such that there is schedule that could turn into F.
- 2.) Consistency: There is no reachable configuration where correct processes decide different values
- 3.) Convergence: For any initial configuration, $lim_{t\to\infty}$ Pr[a correct process has not decided withing t steps] = 0

## 4.5 Lemma 1

**Lemma 1**: Within a k-resilient consensus protocol, for any reachable configuration C and for any subset S of processes that contains at least $n-k$ correct processes either $C_s \vdash F_s^0$ or $C_s \vdash F_s^1$. For a full proof of lemma 1, see Toueg et al. 1985.

## 4.6 Theorem 1: Lower Bound for Fail-Stop Case

**Theorem**: There is no $[n/2]$-resilient consensus protocol for the full-stop case.

**Proof**: Assuming towards contradiction consider a consensus protocol exists such that for $k = (n/2)$ it is k-resilient. From lemma 1 consider a schedule $\sigma^0$ such that for some configuration $C_s \overset{\sigma^0}{\vdash} D_s^0$. It reasonably follows that for $C_{\overline{s}} \overset{\sigma^1}{\vdash} D_{\overline{s}}^1$ since for any schedule of operations, the complement of operations should lead to the opposite conclusion. Note that $D^1$ and $D^0$ were chosen arbitrarily and could be used vice versa in the equations as long as each equation result in differing $D^i$. However, consider if you were to concatenate these two schedules together through an expression such as $\sigma^0 * \sigma^1$, effectively creating a new schedule $\sigma^{0*1}$.

Now consider the two starting configurations $C_s$ and $C_{\overline{s}}$ with this new schedule. It would result in $C_s \overset{\sigma^{0*1}}{\vdash} D_s^1$ and $C_{\overline{s}} \overset{\sigma^{0*1}}{\vdash} D_{\overline{s}}^0$. Note that there would be the opposite conclusions (either from $D_s^0$ to $D_s^1$ or vice versa or $D_{\overline{s}}^0$ to $D_{\overline{s}}^1$ or vice versa) from the initial configurations. However, this leads to a contradiction to the definition of consistency of a k-resilient consensus protocol, thus proving the theorem. □

## 4.7 Theorem 2

**Theorem 2**: For any k, $0 \leq k \leq [(n-1)/2]$ for a standard asynchronous consensus protocol, it is k-resilient for the fail-stop case.

**Proof**: In order to prove that this consensus protocol must exist, we must simply prove that it holds to the three properties of a k-resilient protocol stated above: bivalence, consistency, and convergence. Note that a standard asynchronous consensus protocol is defined as using a fair scheduler, a communication method in which all nodes may talk to each other, whether or not the algorithm is handled through a leader, a counting method that can collect votes from nodes, and follows closely to the pseudo-code for practical Byzantine fault tolerance in that there are phases and communication methods. For an example of such an algorithm, visit figure 1 on page 829 of Asynchronous Consensus and Broadcast Protocols.

Now that we have defined what consensus protocols we will consider, we will start by proving that such an algorithm with $(n-1)/2$ faults holds to our definition of consistency. In order to maintain consistency the protocol must uphold the quorum intersection property. This property states that for any$n$correct nodes, two quorums picked at random $Q_1, Q_2$ will have an intersection of at least 1: $Q_1 \cup Q_2 \geq 1$. Let $q$ represent a quorum size. In order for $Q_1 \cup Q_2 \geq 1$ to hold true, then $q + q > n$ must hold true as well. We can then solve this inequality finding $q = (n+1)/2$. However since we cannot have half a node, we round up to find $q = (n/2) + 1$. Since a quorum is the minimum number of correct nodes needed, then to find the maximum number of faults we can solve through $k = n - ((n/2) + 1)$ finding $k = (n/2) - 1$ in order to maintain

consistency. It reasonably follows that since we have proved an upper bound for $k$, any $k$ less than $k = (n/2) - 1$ will also follow the rule of consistence, proving that for any k, $0 \leq k \leq [(n-1)/2]$ consistency is maintained.

Let us consider the pseudo-code for practical Byzantine fault tolerance. Due to processes waiting on other processes within its protocol, if a process were to experience network failure or fail-stop, this could cause algorithms to hang indefinitely, breaking convergence (see Fischer et al., 1985 for an exploration of the impossibility of asynchronous protocols in the fail-stop case). Importantly, contrary to Fischer et al., in our protocols we will assume that within an infinite time, all nodes will regain function and send a message. This assumption, allows us to assert that as $lim_{t\to\infty}$ Pr[a correct process has not decided withing t steps] = 0 where t is number of time steps. This allows our protocol with our assumption to hold to convergence.

Finally, we must prove bivalence. Let us consider a consensus protocol that has $n$ processes, all with the same input value of 0. All correct processes should decide the same value of 0. If there were a different input value in all $n$ nodes, all correct processes would decide 1. This property proves that our consensus protocol holds to bivalence.

Due to the described standard asynchronous consensus protocol holding to consistency, convergence, and bivalence, when $0 \leq k \leq [(n-1)/2]$ in the fail-stop case we have proven it k-resilient. □

### 4.8 The Fail-Stop Case Concluded

We have now proven that there can only be $k = (n-1)/2$ fail-stop nodes in asynchronous consensus protocols accounting for the fail-stop case. This makes the quorum needed for correct processes to be $q = (n+1)/2$.

## 5 The Malicious Case

### 5.1 Malicious

Consider an asynchronous model of $n$ interconnected processes. These processes will individually be called nodes. These processes have the ability to communicate with each other with messages that tell other processes they are correct, and the value that they wish to contribute. The goal of the nodes is to have all processes agree upon a single value. However, some of these nodes can lie, telling other processes that they are correct, but send an incorrect value. How can algorithms still come to consensus, knowing that some of these processes may exist. They do so by using voting techniques, specifically the requirement that the protocol must reach a quorum of votes in order for the protocol to conclude consensus. In this section, we aim to identify the up-

per bound for how many malicious nodes can occur while a protocol can still continue to function, reaching a consensus on a correct value.

## 5.2 Lemma 2

**Lemma 2**: With a k-resilient consensus protocol, for any reachable configuration C, and for any subset S of processes that contains at least $n - k$ correct processes, either $C_s \vdash F_s^0$ or $C_s \vdash F_s^1$ by some legal schedule where 0 and 1 are the possible binary consensus values. For a full proof of lemma 2, see Toueg et al. 1985.

## 5.3 Theorem 3

**Theorem 3**: There is no [n/3]-resilient consensus protocol for the malicious case.
**Proof**: Assuming towards contradiction consider such a consensus protocol exits where is is resilient to $k$ malicious faults where $k = [n/3]$. Let S and T be subsets of processes of size $2n/3$ such that $|T \cup S| = n$. Note that $|T \cap S| \leq n/3$. Now consider a reachable configuration C. If all malicious processes follow the protocol and continue to follow the protocol, then by lemma 2 $C_S \vdash F_S^i$ and $C_T \vdash F_T^j$ for arbitrary decisions i and j.

Suppose there are legal schedules $\sigma_0$ and $\sigma_1$ such that $C_S \overset{\sigma_0}{\vdash} F_S^0$ and $C_S \overset{\sigma_1}{\vdash} F_S^1$. Consider C, where, by schedule $\sigma_0$, if all processes are correct we reach the decision $F_S^0$. However, if we were to suppose that all processes in $|T \cap S|$ were malicious, then the malicious processes in S would flip their decision creating a configuration $C'_S$. This $C'_S$ differs from $C_S$ only in that $C'_S$ could have additional messages added during the execution of $\sigma_0$. Since $C_S \overset{\sigma_0}{\vdash} C_S^0$, the processes in S that could follow the schedule $\sigma_0$ from configuration C' until all correct processes in S decide 0. Since there is a difference in starting configuration and their decisions are the same, this breaks consistency and is a contradiction. This contradiciont proves the thereom. □

## 5.4 Theorem 4

**Theorem 4**: For any k, $0 \leq k \leq [(n-1)/3]$ a standard asynchronous consensus protocol is k-resilient for the malicious case.
**Proof**: In order to define the type of asynchronous consensus protocol we will make the same assumptions about the protocol as in theorem 2. The only difference will be that this algorithm accounts for malicious nodes.

Similar to theorem 2, in order to prove that such a consensus protocol exists holds for the the three properties of a k-resilient protocol as stated above: bivalence, consistency, and convergence.

In order to prove that an asynchronous consensus protocol with (n-1)/3 faults maintains consistency we will again use the quorum intersection property. Due to this property we can posit that if you were to pick two quorums at random $Q_1$ and $Q_2$ they would have a worst case scenario of $|Q_1 \cap Q_2| \geq 2q+1$ overlapping nodes. This would mean that every faulty node would overlap with a single correct node. We also know that since there is a correct node then $2q+1 > f$ where f is the number of faulty nodes. If you then rearrange this inequality to get $2q > n+f$ and substitute $n$ with the known Byzantine fault node requirement of $3f+1$, you would get $2q > 3f+1+f$. Which can be simplified to $q \geq 2f+1$. Now that we have found the minimum number of correct nodes we need, we find the maximum number of faulty nodes through $k = n-q$ where q is $2f+1$. We find that $k = (n-1)/3$. It reasonably follows that since we have proved an upper bound for $k$ that any $k$ less than $k = (n-1)/3$ will also follow the rule of consistence, proving that any k, $0 \leq k \leq [(n-1)/3]$ consistency is maintained.

The proof of convergence is the same as Theorem 2 in Section 3.7.

The proof of bivalence is the same as Theorem 2 in Section 3.7.

Due to the described standard asynchronous consensus protocol that handle malicious nodes holding to consistency, convergence, and bivalence, when $0 \leq k \leq [(n-1)/3]$ we have proven it k-resilient. □

### 5.5 The Malicious Case Concluded

We have now proven that there can only be $k = (n-1)/3$ fail-stop nodes in asynchronous consensus protocols accounting for malicious nodes. This makes the quorum needed for correct processes to be $q = 2f+1$.

## 6 Applying Proofs to Implemented Algorithms

### 6.1 Algorithms

It is extremely difficult to prove k-resilience for real-world applications of consensus protocols due to the increased complexity and personalization to the distributed systems they are written for. There is also variability across real-world implementations of consensus algorithms causing a proof for one algorithm to be useless for another. However, the proofs above have been written for generalized consensus algorithms. In order to apply such proofs to more specific algorithms we must define the conditions a generalized consensus algorithm holds.

### 6.2 Generalized Stop-Fail Algorithms

For the fail-stop case a consensus algorithm must be an asynchronous system of $n$ fully interconnected processes. These processes must communicate

through messages organized with a message system. This message system will send and receive function where there is a buffer between the two functions. If a send function is called, it will instantaneously place a message m in the process's buffer. If a receive function is called, it will remove some message from the buffer for the process calling the receive function.

The consensus protocol must operate in atomic steps, as defined in Section 3.2. With the culmination of these atomic steps making a schedule that allows for a configuration to change. This scheduler must be a fair scheduler, as defined in Section 3.3.

Finally, the consensus protocol must have a system of voting. A node or nodes must collect messages and create a decision by some type of majority based on values from the messages. This technique of voting is dependent on the protocol at hand, however, the system must have a way to verify that decisions had more than a given or calculated number of results.

Fail-stop nodes will simply not send messages to other nodes. The method of making a node fail-stop does not matter, as long as other nodes do not recieve the message.

### 6.3 Generalized Malicious Algorithms

The generalized algorithm that accounts for malicious nodes is the same as the generalized fail-stop algorithm with several additions. There must be an identifying way for nodes to tell if messages from other nodes can be correct (i.e. hashing). There will also be malicious nodes which can send false and/or contradictory messages, can fail to send messages, and can change its internal state to any other state.

### 6.4 Leaderless Byzantine Paxos

Due to leaderless byzantine paxos being apart of the paxos algorithm family, the proof for its correctness with k-resilience has been widely published. For a full proof we recommend Lamport et al. 2011.

### 6.5 Raft

Due to Raft being a popularized industry choice for consensus algorithms, there is a formal proof for its correctness with k-resilience. For a full proof we recommend Ousterhout et al. 2014.

### 6.6 HotStuff

For this proof we will be considering the pseudo-code for HotStuff consensus algorithm, on page 7 from *HotStuff: BFT Consensus in the Lens of Blockchain*,

Abraham et al. 2019. We will specifically be focusing on Algorithm 2. In order to prove that the quorums will still apply for this algorithm, we simply must prove that the pseoudocode fits with our previously defined generalized malicious algorithm.

Since the algorithm is iterating over all nodes, and there is no synchronous properties that appear within the code, it is evident that the consensus protocol is asynchronous and works between $n$ processes. The communication method is established in each phase. Looking at lines 6 and 8 in the pseudo-code, broadcast and wait are methods that act as send and receive.

Additionally, it can be seen that each process as well as the overall protocol iterates in atomic steps. As a reminder, atomic steps are defined as a process trying to receive a message, perform an arbitrary computation, or sending a finite set of messages. Each operation performed within the entire pseudo-code is one of these processes. The most common operations are send and broadcast, whihc as established ealier, operate as send or recieve, matching the definition of an atomic step. Other operations simply call functions. However, looking at the function, they only hold send or broadcast operations. This limits the functions of the pseudo-code to be atomic steps.

We must also consider whether the consensus protocol follows a fair scheduler. Let us consider $R(q, p, t)$ which represents the even that p receives a message from q in round t. We must show that there is a positive constant $\epsilon$ such that $Pr[R(q, p, t)] > \epsilon$. What this means, is that the probability of such an event finishing, is greater than 0. Recall that, for the purpose of this paper, we are making the assumption that any faulty processes, if given infinite time steps, will regain function, as stated in Section 3.7. This would ensure that the event $R(q, p, t)$ would occur making $\epsilon$ a positive integer greater than zero. The second property a fair scheduler must hold is that for r, p, q, and round t, the events $R(q, r, t)$ and $R(q, p, t)$ are independent. As seen in the pseudo-code each view, or iteration of the algorithm, there is a leader. This leader will consider messages from each follower node. Upon receiving each message, they will be processed independently from each other as seen in the definition of broadcast and wait in Section 3 of *HotStuff: BFT Consensus in the Lens of Blockchain*, Abraham et al. 2019. This ensure that each event $R(q, r, t)$ and $R(q, p, t)$ will be considered independently in the algorithm, proving that the HotStuff consensus algorithm does use a fair scheduler.

In order to prove that the consensus protocol contains a type of voting system, we must only look at lines 27-34 which outlines a system of changing followers' messages. After going through three phases, in which the leader node collects $(n - f)$ messages from $(n - f)$ nodes, it ensures that the leader has the correct message. After such assurance, the leader then broadcasts this correct message to all followers, having them change their message to the new, correct message. Thus, the algorithm has a voting method, further ensuring the algo-

rithm fits to our generalized malicious algorithm definition.

Finally, we must consider whether the algorithm accounts for malicious nodes, and contains an identification method for communicating correctness. While in this pseudo-code excerpt, there is no clear representation of a communication of correctness function, we refer to subsection of Section 3: Cryptographic primitives in *HotStuff: BFT Consensus in the Lens of Blockchain*, Abraham et al. 2019. In which, a type of hashing is identified that would allow for leader nodes to identify correct nodes through the quorum certificates. This allows for us to conlude that communication of correctness is possible.

To prove the algorithm handles malicious nodes, we refer back to the pseudocode. At each phase of the algorithm, there is a wait function that waits for $(n - f)$ messages. As long as this voting technique works, and $f$ has been calculated correctly, then the algorithm must have the ability to account for malicious nodes, due to its ability to wait for $(n - f)$ messages.

Since we have shown that the HotStuff consensus algorithm fits with out generalized malicious algorithm definition, it follows that the proof for the upper bound of $k$ for consensus protocols that account for malicious nodes holds for this algorithm. □

# 7 Overview of Integrative Exercise

This paper was written for the integrative exercise, consensus in a faulty system at Carleton College. This project was advised by Professor Tanya Amert. The students involved were Colin James '25, Arisha Khan '25, Wesley Yang '25, Vanessa Heynes '25, Luha Yang '26, and Hanane Akeel '25.

In our integrative exercise, consensus in a faulty system, we aimed to learn about consensus protocol, implement different types of consensus protocols in a real-world application, compare protocols in terms of efficiency and speed, and show proofs of k-resilience for previously unproven algorithms.

In order to research consensus algorithms, we first had to understand what consensus algorithms were and how they were used. In order to do so we used the Byzantine Generals Problem as a model for consensus and nodes. The Byzantine generals problem can be visualized as a game with $n$ generals and a centralized Rome (or other empire the generals can take over). Generals can decide to attack or retreat, with no consequence to either, however, the goal is for all of the $n$ generals to decide the same thing. In order to do so, generals communicate with each other, making sure that all generals will agree to the same decision. This problem seems relatively easy. However, what if generals could die or lie within their communications making it more difficult for generals to make their decisions. This is where consensus algorithms come in, acting as

a way to ensure that consensus among the generals will be reached.

After understanding a model for our consensus algorithms, we then had to understand the basics of consensus algorithms. This included researching the differences between leaderless and leader-based algorithms as well as synchronous and asynchronous. We also had to understand different types of faults, ensuring that we could implement them in our code. After this basic understanding, we were then able to move onto looking at actual consensus algorithms.

The first consensus algorithm we looked at was Practical Byzantine Fault Tolerance (PBFT). This was among the first published consensus algorithm and provided us with valuable knowledge about the complexity of the consensus problem we were solving. It introduced the idea of a multi-phase system that allowed for multiple communications between leaders and followers. Understanding this concept made the other algorithms much easier to implement and understand.

Moving on from PBFT, we sought to understand the differences in implementation of leaderless and leader-based consensus algorithms. To do so, we split into two groups of three with one group working a leaderless consensus algorithm and the other working on a leader-based algorithm. The leaderless consensus algorithm chose to implement Leaderless Byzantine Paxos and the leader-based group implemented RAFT. During these implementations we ran into our first challenges.

We needed a way to organize communication between multiple nodes, with a cluster of raspberry pi's acting as our distributed system. The solution was a an open source library called openMPI. This library allowed us to facilitate communication between all nodes of our distributed system. However, using openMPI required research and understanding of both the library and our distributed system.

Once we understood consensus algorithms, communication methods, our distributed system, and the algorithms we wanted to implement, we were finally able to start implementing the algorithms. After the successful implementation of Leaderless Byzantine Paxos and RAFT, we then debated what the next best consensus algorithm was to implement. We decided on a leader-based consensus algorithm called HotStuff.

HotStuff, published in 2018, is a leader-based consensus algorithm that offers several aspects other algorithms could not. It has linearly view change, meaning the process of switching leadership requires linearly scaling communication complexity, as well as responsiveness, meaning latency is based on network delays rather than average time bounds. Due to these benefits, we decided to implement HotStuff over other possible consensus algorithms.

Once we had successfully implemented HotStuff, we decided that an additional algorithm making improvements on HotStuff or a different algorithm would be impossible in the time we had left. Instead, we focused on making HotStuff more efficient and easier to work with. We also wrote a proof of HotStuff being k-resilient that had not previously existed, or at least not in the same technique we proved it in.

Overall, during our capstone project, our group has researched consensus algorithms, implemented three consensus algorithms used in the real-world, researched consensus algorithm theory, and proven HotStuff's k-resilience.

# References


[1] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness.

[2] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[3] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. J. ACM 32, 4 (Oct. 1985), 824–840. https://doi.org/10.1145/4221.214134

[4] Yin, Maofan et al. "HotStuff: BFT Consensus in the Lens of Blockchain." arXiv: Distributed, Parallel, and Cluster Computing (2018): n. pag.

[5] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," Journal of the ACM (JACM), vol. 32, no. 4, pp. 824–840, Oct. 1985, doi: