A Brief Study on Byzantine Fault Tolerance, Consensus, and Blockchain

Zhiping Li, Hanmin Zhong, Bo Zhang *, Ruolong Ma

CAICT, Beijing, China

* Corresponding Author Email: zhangbo3@caict.ac.cn

Abstract. Distributed systems solve large scale complex problems and consensus is at its core, coordinating every single subsystem towards the same goal. Among consensuses, byzantine fault tolerating consensus is a particularly useful kind of consensus which we will extensively discuss. In this paper, we will review the general approaches to solve the consensus problem – deterministic consensus and probabilistic consensus, PBFT and HotStuff's frameworks and properties, impact that HotStuff has on consensus' framework in the future, and how federated learning uses principle of distributed system to leverage privacy.

Keywords: Consensus, distributed system, byzantine fault tolerance, blockchain.

1. Introduction

Why do we need a distributed system? I think that this question ought to be answered before starting any discussion about it, and truth be told, we don't usually do. Today's computer hardware offers high performance, one can easily run websites on their own computers or train models with hundreds of gigabytes of data. The computing power offered by modern hardware is almost overkill for personal usages, so monolithic systems cover needs for most situations, but for complex cases, one single powerful system is far from enough. Take Platform-as-a-Service as an example, if users on two continents use the same service, then it's expected for them to be able to use service of equivalent quality, so it's natural to deploy server in different continents to compensate for latency, and many implicit consequences cometh with, for example, if the PaaS we are talking about is an online ticket service, and two users try to buy the last ticket of a movie's premiere, then the ticket should only be sold once, so it's the system's duty to make sure that both users are treated equally and see the same kind of information. To achieve this behavior, the system must overcome factors such as internet latency, system loads... to reach agreement with each other. The resolution of such a problem is a consensus protocol that coordinates each subsystem to reach equal states so users stay on the same page, and there is more to it, the system should provide consistent and reliable service so it can be used at any time, and even when someone tries to break the system, it should detect such malicious behavior and has countermeasure. The above properties that a distributed system should have been concluded as per **CAP** theorem: availability, consistency, and fault tolerance, as shown in Fig. 1. We will talk about different kinds of consensuses in the next section and some implications of the **CAP** theorem.

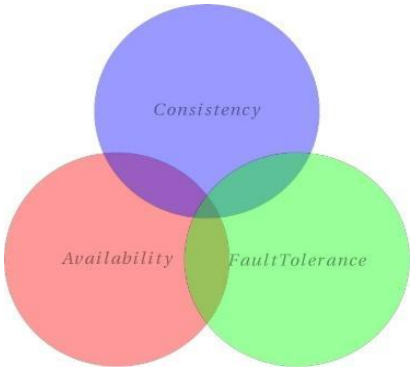

Figure 1. CAP theorem

2. Consensus Nowadays

2.1. Deterministic Consensus

When a system doesn't rely on any kind of probabilistic method in making decisions, then this system can be called deterministic. To explain in another perspective, given input, a deterministic system always outputs the same. This property makes testing the correctness of system very simple: no matter the complexity of the system, the output can always be predicted. When a problem occurs, trial-and-error almost always indicates where the error is. A deterministic system, however, is plagued by the scaling problem, most deterministic systems need a $\mathbf{O}(n^2)$ complexity to convey messages among systems. There exist systems allowing linear complexity message communication as well, at cost of network delay.

(1) Ideal Consensus

In the most basic distributed system, we can treat each system as an independent process, each process has a queue of commands, and the goal of the whole system is to keep these processes' commands the same. The most direct way to keep these systems in the same state is using a point-to-point communication after every single action, as a naive implementation of two-step commit solves the problem. There is, however, a premise that we need for two-step commit to work: the network will never go haywire, and systems will not deliberately delay or tame messages, and this is only possible under ideal circumstance. By referring to being ideal, every single node inside system is always reachable within an expected limit of time, no node is acting maliciously on its own, and the actions performed are always done correctly. This behavior is the goal that all distributed system tries to reach via the help of consensus. The problem we are trying to solve is difficult in nature, but the smart design of consensus solves some seemingly impossible problems.

(2) Consensus in Reality

Network is complex, and any simple mistake make cause consensus to fail, so much is taken into consideration when designing a consensus protocol, we can classify these problems into 3 categories: single-point failure, network delay, and faulty node. One thing to note is that the three categories above merely cover a small percentage of possible errors, many errors of more subtle nature exist, but these three should be accountable of their cause.

Single-point failure: when there is a hub that transfers messages among systems, then it's most likely to be the source of any single-point failure. To amend this problem, we need redundancy, when failure happens, we need to replace the old hub with a new one. This is one of those *easier said than done* scenarios, because what we are trying to achieve is reaching consensus when consensus fails. The fault tolerance of system is crucial to solve this problem. **PBFT** [1] provides the full process of changing the hub (we would later refer it as primary), but the complexity may rise to $\mathbf{O}(n^3)$. Almost all post-PBFT consensuses have similar structures.

Network delay: unlike single-point failure that might or might not be an issue, network delay almost always presents. Since this problem can't be avoided, a workaround should always be included in the design. There are 3 different kinds of internet delay that is of our interest [2]: synchronous, partial synchronous, and asynchronous. In a synchronous network, there a specific upper limit, Δ, that all nodes must follow. A synchronous system is always consistent and available. In an asynchronous network, there is no limit. Any node can delay for a finite amount of time. The only guarantee is that messages will be delivered, eventually. The cause of such system roots from the complexity of the internet. An asynchronous system can never reach consensus, even in face of one single faulty node [2]. This strong conclusion is a theoretical explanation to the limit of asynchronous system, since otherwise, there can be no communication in internet at all. The impossibility to reach consensus and forming a consistent system can be circumvented by assuming asynchrony to some degree, and this is where partial synchrony arises. In a partial synchronous network, there is a special point of time called Global Stabilization Time, **GST**, where network is considered asynchronous before it, but synchronous after it. **GST** exists, but when it occurs is unknown, and at any time x, the message will arrive at $\Delta + max\ (x, \mathbf{GST})$ latest. Partial synchronous seems not practical at first sight, but in practice,

it's a good compromise between synchronous and asynchronous network. For synchronous network, the Δ determines how the network behaves, but a constant period of wait time is not useful, when it's set too small, timeout happens too often, when its set too large, the delay becomes unbearable. Obtaining a perfect value takes too many trials and is unlikely to fit all nodes. To obtain a better value, we should increase or decrease dynamically to obtain safety of the consensus protocol under network fluctuations.

Faulty nodes: faulty nodes are so destructive to the system that there are two kinds of consensuses, one that tolerate byzantine faults, and one that doesn't. Byzantine fault can be understood as nodes making arbitrary actions, it might not follow instructions sent to it, or even organize attack according to some scheme. This aspect represents exactly the fault tolerance in CAP theorem. Tolerating byzantine faults means a degradation in performance, as protocols that tolerate $\lfloor x/2 \rfloor$ faulty nodes can now only tolerate $\lfloor x/3 \rfloor$ faulty nodes. This is however a premise that must be strictly followed by the system, or there can be arbitrary behavior happening.

Concluding the three categories of problems sheds light into understanding why modern BFT protocols' components. Single-point failure is solved with view-change, network delay is taken care by considering different models of network, and faulty nodes are considered as part of the system itself. Post-PBFT BFT consensus protocols share similar procedures with PBFT while optimizing each single step, and more subtle details to improve consensus efficiency.

One would argue that CAP theorem doesn't fit into today's systems well [3], indeed, the systems we talk about are several degrees more complex than simple systems that read-write one transaction at a time, so is the scale of system. Take availability from CAP theorem for example, it means that every request from non-faulty node must be responded correctly, but in modern systems, availability mostly equals uptime of system, but by borrowing terms from CAP theorem, one could oversee the underpinning of modern systems in a much more simplistic view, with a few modifications of the original CAP theorem to adapt for modern system.

2.2. Probabilistic Consensus

Deterministic consensus provides consistency, fault tolerance, and gives a reliable upper bound estimate, these properties make deterministic consensus widely used in distributed systems. One can notice that the most widely used distributed systems are cluster of servers managed by a centralized organization, and user doesn't control over their own data and privacy. To make decentralized distributed system, one must come up with new consensus because of the scaling problem that increase exponentially. Probabilistic consensus is used to solve this problem. In addition, probabilistic consensus provides higher availability than deterministic consensus because of its nature of being incentive-driven and higher decentralization. Theoretically, the possibility that probabilistic consensus goes wrong is zero, but attacks on probabilistic consensus had succeeded and various method to exploit it exist.

(1) Proof-of-Work

Deterministic consensus provides consistency, fault tolerance, and gives a reliable upper bound estimate, these properties make deterministic consensus widely used in distributed systems. One can notice that the most widely used distributed systems are cluster of servers managed by a centralized organization, and user doesn't control over their own data and privacy. To make decentralized distributed system, one must come up with new consensus because of the scaling problem that increases exponentially. Probabilistic consensus is used to solve this problem. Theoretically, the possibility that probabilistic consensus goes wrong is zero, but attacks on probabilistic consensus had succeeded and various method to exploit it exist.

(2) Proof-of-Stake

PoW's problem lies in its intrinsic nature: computation power dominates right of producing blocks. To improve **PoW** consensus, one needs to find another method to replace computing power, and this is where stake comes into play. Compared to **PoW**, **PoS** doesn't require as much hardware, what

replaces it is stake, simply put, you are putting some sort of currency into it. By replacing hardware with stake, cost of computation is largely reduced, and many more aspects are improved.

(3) High Cost of Attack

When damage is done to a **PoW** system, users can fork voluntarily to prevent further damage, but since **PoW** uses fixed algorithm to formulate puzzle, **ASIC** machines can launch attack repeatedly, only losing electricity cost but always could gain reward for the next block. To prevent this nothing-at-stake scenario from happening, **PoS** ensures that users have stake on decisions they make. For example, when a fork happens, nodes would know those who vote on other heights, therefore eradicating the malicious nodes' stakes.

(4) Diverse choice of consensus

PoS can use two kinds of consensus: chain-based **PoS** and BFT-based **PoS**. In theory, **BFT**-based **PoS** is safer, because chain-based **PoS** is decided several heights in advance.

Table 1. Comparison of Common Properties among Consensuses

	Deterministic Consensus		Probabilistic Consensus		
	PBFT	HotStuff	PoW	PoS	Delegated PoS
			Bitcoin	Casper	DRBFT
Safety	Yes	Yes	Mathematically approaches probability of 1	same as PoW	same as PoW
Liveness	Yes	Yes	Yes	Yes	Yes
Efficiency	High with small number of nodes	High with a few thousands of nodes	Consumes lots of energy [4]	Consumes less, but still involves PoW-like method	Same as the underlying consensus protocol
Scalability	Significant performance degradation beyond 20 nodes	Scalable without performance compromise	Scalable at the cost of energy consumption	Easily scalable to thousands of nodes [5]	Same as the underlying consensus protocol

In table 1, I included comparisons of both deterministic consensuses and probabilistic consensuses in term of their safety, liveness, efficiency, and scalability.

3. Hotstuff – a new framework for consensus

In PBFT is the first consensus algorithm that correctly handles byzantine faults at asynchrony. Since then, algorithms follow a design pattern similar to that of PBFT's to tolerate byzantine faults, including message transfer, view change, and consensus algorithm. **HotStuff** proposes a new paradigm to make consensus more efficient while still maintaining safety and liveness in partial synchrony.

3.1. Brief review of PBFT and HotStuff

A general paradigm to reach consensus stems from two-phase commit, while effective, it's limited to situations without primary node failing or non-primary nodes behaving maliciously, and three-

phase commit solves the above problem. The following section serves as a brief review of PBFT and HotStuff consensuses which is not relevant to the rest of the paper, so if you are already familiar with them, you can skip it.

(1) PBFT

PBFT follow a three-phase paradigm consisting of pre-prepare, prepare, and commit. In pre-prepare phase, primary node multicasts the pre-prepare message to other nodes to notify sequence number, view number, and request from client. In prepare phase, if a node accepts pre-prepare, it multicasts prepare to all other nodes to notify that it is ready to execute the request. In commit phase, if a node accepts prepare, it multicasts commit to all other nodes to notify that it receives from all other nodes, then executes the request from client. Accepting message from another node depends on view number, sequence number, and node states, when a threshold of accepted messages is reached, the node proceeds to the next phase.

(2) HotStuff

HotStuff adds an additional phase, incorporating view change prior to every round of consensus. On top the extra phase, **HotStuff** uses threshold signature in contrast to PBFT's point-to-point message communication. During each phase, non-primary nodes only send messages primary node, and when primary collects messages reaching the threshold amount, it would encompass them into one single message and send it to non-primary node.

3.2. HotStuff's innovation

Modern system usually starts with a common framework, then improve upon it by optimizing each part. Take distributed system for example, **PBFT** first gave the blueprint of how BFT consensus works around a 3-commit phase. Consensus after revolves around the same principle: **Aardvark** [6], **Zyzzyva** [7], **RBFT** [8] … All consensuses must fulfill two properties simultaneously: safety and liveness, the nodes communicate with each other to make sure their states are up to date. This is the fundamental reason of system's complexity, the amount of messages exchanged scales fast. Whenever a primary node misbehaves, a view-change happens, and if cascading errors occur, the complexity rises further. To solve this issue, we can cast safety away from liveness, and design them as individual modules. **HotStuff** detaches liveness from safety in the protocol. During any normal process, **HotStuff** always gurantees safety, furthermore, **HotStuff** combines view-chagne as part of a normal process so that when a view-change happens, complexity would not further increase. As for liveness, **HotStuff** extracted it into a module called Pacemaker. Pacemaker can be arbitrarily designed, such as using a robin-round to select next primary node, or using the method from **DLS** [3], setting a large limit, then decrease it towards threshold. By separating components in a consensus into modules, we can design each part individually. As described in **DRBFT** [9], the delegation took place first, then by using quorum chosen during the delegation step, we use PBFT to reach consensus.

HotStuff's two most remarkable innovations are: making view-change part of the consensus, separating liveness and safety.

(1) View-change

In PBFT, view-change happens only when the primary node shows malicious behavior. When an error is detected, there is at least **f+1** correct nodes sending view-change requests, in which each request will include one **pre-prepare** and its corresponding **2f prepare** messages. When the next primary receives **2f view-change**, the next round of consensus will start, making a full **view-change**'s complexity $\mathbf{O}(n^3)$. But if the following **f-1** primary nodes chosen by the robin-round fashion are also faulty nodes, then **view-change**'s complexity would increase to $\mathbf{O}(n^4)$.

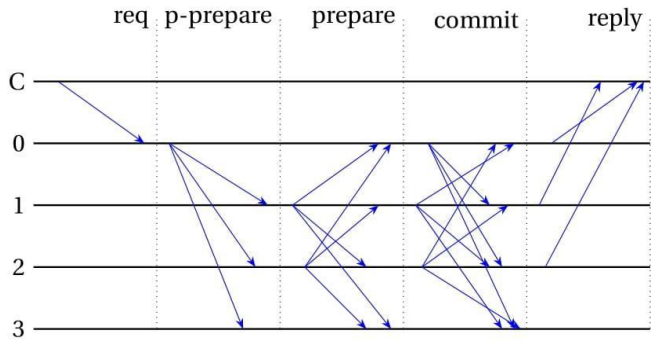

Figure 2. Normal PBFT process

In **HotStuff** however [10], since view-change happens every single round using threshold signature, its complexity is **O**(n), and when cascading failures happen, it would go up to **O**(fn), which is several orders of magnitude less than **PBFT**. In "Fig. 2", we can see that **PBFT**'s message transfers include an all-to-all message transfer, which is the source of the high message complexity. In "Fig. 3", **HotStuff** uses primary node to collect quorum certificate first, then uses threshold signature to send one-to-all messages. This only incurs linear complexity. Making much more efficient usage of the network.

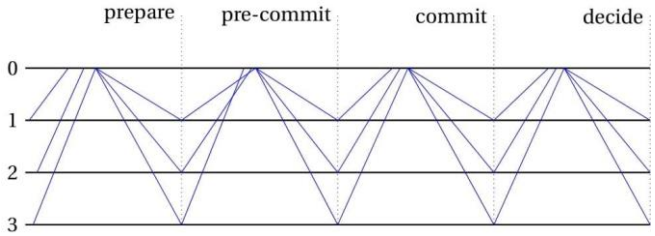

Figure 3. Normal HotStuff process

(2) Liveness

In PBFT, liveness is preserved by 2 methods. First, doubling the wait time if the new primary cannot receive messages. The reason behind is to prevent **view-change** from happening too often because of the complex network topology. Second, there can be at most **f** faulty nodes, so we can guarantee there won't be more than **f** view changes consecutively lead by faulty nodes.

(3) HotStuff at its Core

HotStuff broke new ground for BFT consensus in three ways: linear complexity for message communication, parallel consensus, and separating liveness as a module.

The most prominent difference of **HotStuff** is the complexity of each message communication decreases from **O**(n^2) to **O**(n). As per **PBFT**, when a message is communicated, each node sends a message to every other node, so there can be as much as n^2 messages existing simultaneously in the network. In this way, after each message communication, every node would become aware of every other node's state. The con is the quadratic increase in complexity as number of nodes increase. In practice, when total number of nodes is beyond 20, **PBFT**'s performance suffers significantly. To mitigate the performance degradation that is caused by the scaling problem, **HotStuff** uses a new way to communicate between nodes. When a message is transferred, non-primary nodes only send to the primary node and receive from the primary. Primary node sends to all other nodes and receive from all other nodes. The primary node gathers messages from other nodes and conglomerate them using threshold signature to compose them into one single message, in return, there can be at most n messages simultaneously while achieving the same effect as before. The tradeoff is the two-way travel time compared to the previous one-way travel time. The time is doubled in number, but in real practice, it's not, for the following reason. In a network, as number of nodes increase, time to transfer messages doesn't only increase linearly as the number grows, but more likely to be exponential. So, in a traditional BFT consensus, as number of nodes increase, the time for messages to transferred increase exponentially as well. However, as **HotStuff**'s complexity only increases linearly, when the same

number of nodes is added into the system, time taken by messages to be transferred is much less compared to the traditional case.

During a round of consensus, nodes simply exchange messages, receiving messages, and verifying messages. And the repetition of these steps is carried out linearly. To avoid this redundancy, **HotStuff** uses messages with identical structure, so there is no need to collect messages when the same set of threshold signatures can be used for different steps at the same time. As shown in Fig. 4, we can see that the same certificate is used throughout cmd1 to cmd4, surpassing certificate efficiency in Fig. 5, which is used once in each command and then re-fetched during the next round of consensus.

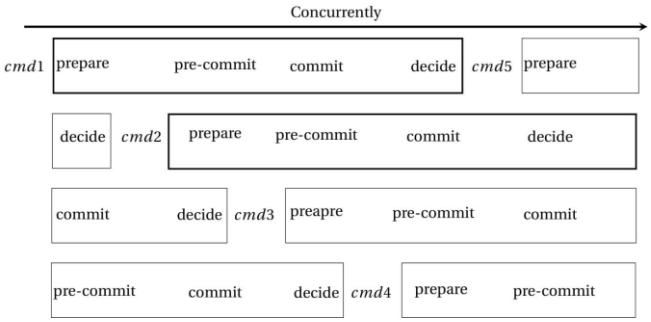

Figure 4. HotStuff uses same quorum certificate for different rounds of consensuses

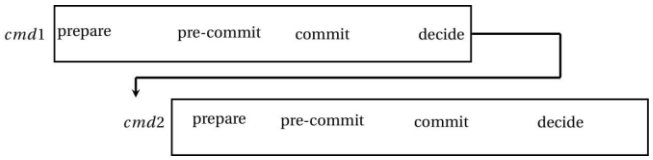

Figure 5. PBFT recollects quorum certificate for every single step

4. Leveraging privacy with distributed system

Thus far, we see that distributed systems are powerful tools that can be deployed in a scalable fashion. Its usage can be, but not limited to, a conglomerate of servers managed in a centralized manner, incentive-driven individuals forming a large, decentralized network... These schemes coordinate a large and complex system to work towards a same goal. By deploying a distributed system, one can have more availability, scalability, and efficiency, but there is more to it, privacy.

The training process in machine learning usually take place in a server with all training resources and data stored locally. This might result in leak of private information when the server is compromised. Implicit dangers also exist within such training paradigm. Attacker might infer from results the underlying subjects, information obtained can be as specific as individual's clinical or social security information, and such concern is about differential privacy.

4.1. Data Privacy

Borrowing from distributed system, federated learning achieves model privacy. The training method can be concluded as following: the primary node sends to all other nodes the current model, non-primary node makes computation individually, updates the model, might be an update using gradient descent, and sends the update back to the main node, main nodes update model based on feedbacks from non-primary nodes. We can see that the main node doesn't store any data, and computational detail remain within non-primary nodes, so there is no risk of data compromise in this training model.

4.2. Differential Privacy

Biomedical engineering is a rising industry that uses machine learning extensively. Data used in BME involves sensible detail such as name, age, social security number, pathology record... and we would like these details not to be disclosed in the final model, so privacy would be at risk. But there

is more to it. [11] shows the possibility of extracting the entire model by a query adversary. Adding randomness into model mitigates this issue by changing a deterministic behavior of choosing parameters into probabilistic. [12] In the most ideal case, when one sample is changed inside a set, there is no significant change in model that can be used to infer the underlying parameters.

5. Conclusion

Distributed system makes it possible to deploy and use the network at a much larger scale. Different scenarios and demands advance the developments of distributed system, and the design and implementation of more efficient consensus is at the core of the developments. **TCP/IP** sheds light into how to use one-to-one message to reach agreement, which is one of the first tries to solve the problem of consensus. **PBFT** constructs a framework for almost all consensuses, making a fault tolerance system practical. For the present and the foreseeable future, web3 provides the idea of a new internet environment, and to implement this trust-based environment, distributed systems and consensus mechanisms are innovated and optimized.

6. Acknowledgments

This work was financially supported in part by the 2020 Industrial Internet Innovation and Development Project: Network Identifier Construction Project.

References

[1] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in Proceedings of the Third Symposium on Operating Systems Design and Implementation, 1999.

[2] M. Fischer, N. Lynch, and M. Paterson, “Impossibility of Distributed System with One Faulty Process,” Journal of the Association for Computing Machinery, 1985.

[3] C. Dwork and N. Lynch, “Consensus in the presence of partial synchrony,” Journal of the Association for Computing Machinery, 1984.

[4] Digiconomist, “Bitcoin Energy Consumption Index,” https://digiconomist.net/bitcoin-energy-consumption.

[5] Ethereum, “Upgrading Ethereum to radical new heights,” https://ethereum.org/en/upgrades/.

[6] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, “Making byzantine fault tolerant systems tolerate byzantine faults,” in Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI’09, (USA), p. 153-168, USENIX Association, 2009.

[7] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative byzantine fault tolerance,” ACM Trans. Comput. Syst., vol. 27, jan 2010.

[8] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, “Rbft: Redundant byzantine fault tolerance,” in 2013 IEEE 33rd International Conference on Distributed Computing Systems, pp. 297–306, 2013.

[9] Y. Zhan, B. Wang, R. Lu, and Y. Yu, “Drbft: Delegated randomization byzantine fault tolerance consensus protocol for blockchains,” Information Sciences, vol. 559, 01 2021.

[10] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hotstuff: Bft consensus in the lens of blockchain,” 2018.

[11] F. Tramèr, F. Zhang, Ari Juels, M. Reiter, and T. Ristenpart, “Stealing Machine Learning Models via Prediction APIs”, in Proceedings of the 25th USENIX Security Symposium, 2016.

[12] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith, “Calibrating noise to sensitivity in private data analysis”, In Proceedings of the Third conference on Theory of Cryptography, 2006.