Improving the Communication Complexity and Latency of Byzantine Vector Consensus

Manos Chatzakis*
EPFL
Lausanne, Switzerland
emmanouil.chatzakis@epfl.ch

Jovan Komatovic†
EPFL
Lausanne, Switzerland
jovan.komatovic@epfl.ch

Rachid Guerraoui‡
EPFL
Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Recently, it has been proven that the Dolev-Reischuk bound, which states that any Byzantine consensus algorithm has at least quadratic communication complexity, is tight in partial synchrony, where message delays are unbounded for a certain period of time. Specifically, QUAD protocol has been proposed, which achieves quadratic communication complexity and linear latency for Byzantine consensus in a partially synchronous environment. However, regarding Byzantine vector consensus, where processes need to agree on a vector composed of multiple proposals of distinct processes, QUAD leads to scenarios with cubic communication complexity, while other approaches and alternatives that managed to achieve sub-cubic complexity are impractical due to their exponential latency.

This paper presents a Byzantine vector consensus algorithm that uses a novel leader-based protocol that allows the processes to disseminate their vector to the rest of the system efficiently, namely leader-based vector dissemination, enabling the processes to agree on a hash of a vector, and then to reconstruct it to achieve consensus. Our analysis demonstrates that with our protocol , Byzantine vector consensus is achieved with sub-cubic communication complexity and sub-quadratic latency.

Keywords. Distributed Algorithms, Byzantine Vector Consensus, Partial Synchrony, Leader-Based Protocols, Vector Dissemination

1 Introduction

Motivation. Byzantine vector consensus [10] is a fundamental notion in distributed computing, with applications in a variety of fields, such as blockchain ([11], [9]) and decentralized computing [8]. Intuitively, Byzantine vector consensus requires a system of n processes to agree on a single vector of $n - f$ values, where each value is a proposal of a process. The system can sustain up to f Byzantine processes, i.e. processes that may behave in an arbitrary or adversarial way. We operate in a partial synchronous environment, where the message delays are initially unbounded, until a global stabilization time GST. After GST, message delays are bounded by a known delay δ. Recently, the Dolev-Reischuk bound [4], which states that any Byzantine consensus protocol is at least $\Omega(n^2)$, has been extended to partial synchrony [6], through a protocol that achieves consensus in $O(n^2)$, namely QUAD [1], proving that the bound is tight even in this setting.

Existing solutions cannot work. Protocols such as QUAD, which also achieves $O(f)$ latency, can easily be adapted to support Byzantine vector consensus, but unfortunately due to the $O(n)$ size of the

*Main Author
†Main Supervisor
‡Supervisor

exchanged messages, which is a direct consequence of the fact that messages contain $n-f$ proposals, the resulting protocol can achieve vector consensus in $O(n^3)$. Ongoing research in this direction developed an algorithm that is able to achieve vector consensus in $O(n^2 \log n)$, by addressing the problem of QUAD for messages that contain vectors [2]. Specifically, it exploits a module named vector dissemination, which allows the process to disseminate a vector to the rest of the system, and eventually every process obtains a hash and a proof of a vector that at least $f+1$ process has cached in their memory. This way, the processes use QUAD to agree only on the Hash_Value and Storage_Proof of a vector, which have a constant size, maintaining the quadratic complexity of QUAD. After reaching consensus, the processes reconstruct the final vector based on a hash, achieving vector consensus. Unfortunately, the proposed vector dissemination module of the algorithm has exponential latency $O(n^f)$, which makes its application highly impractical in real-world scenarios.

Our Approach. Our work addresses the problem of the exponential latency of vector dissemination for the vector consensus algorithm of [2]. Specifically, we propose a Byzantine vector consensus algorithm that uses a novel vector dissemination protocol to disseminate the vector of $n-f$ values to the system in order to obtain the hash and the proof as described above. Our approach is able to achieve Byzantine vector consensus with $O(n^2\sqrt{n})$ communication complexity and $O(n\sqrt{n})$ latency. The outline of our consensus algorithm is the following: Each process starts by broadcasting its proposal value to the rest of the system. When a process delivers $n-f$ messages from distinct processes, it creates a vector containing the delivered values and initiates our novel vector dissemination protocol. Eventually, each correct process will obtain a Hash_Value of a vector, and a Storage_Proof proving the validity of the vector. The hash values are then proposed to QUAD, to reach consensus for a single Hash_Value. When each process obtains the decided Hash_Value of the vector, we initiate an asynchronous data dissemination protocol to recreate the vector in each process.

Our vector dissemination protocol is leader-based: it operates in views, where each view has a designated leader, responsible to disseminate its own vector of $n-f$ values to the rest of the system. The processes advance through views till they reach a view with a correct leader, and they stay there for a sufficient amount of time to exchange the needed messages. The views are organized into epochs, where each epoch contains a predefined number of views, $|VE|$. As we show in our analysis in the latter sections, the value of $|VE|$ has a significant role in the communication complexity and latency of the protocol. We implement the novel vector dissemination using two crucial modules: The View Core and View Synchronizer, which are the main ingredient in a variety of leader-based protocols. View Core is responsible for the actions of the processes within a view, while the View Synchronizer is responsible for bringing the processes to a view with a correct leader for a sufficient amount of time. We can approximate the communication complexity of Leader-Based protocols that operate using View Cores and View Synchronizers using:

$$n \cdot C + S$$

where n is the number of processes, C is the maximum number of bits a process sends while executing the View Core, starting from GST till the time the vector dissemination completes, while S is the communication complexity of the View Synchronizer during the same period of time. During each view, where each process executes the steps of the View Core, the leader broadcast its proposal to the rest processes and waits till it gets enough acknowledgments from $n-f$ distinct processes, which is the indication that sufficient processes have stored the message. When a process delivers a vector from the leader, it caches the vector and sends to the leader a signed acknowledgment message containing the Hash_Value of the vector. When the leader gathers enough messages, it composes the signatures into a single threshold

signature and broadcasts that a vector Hash_Value has been decided. Every process that receives this decision message checks if the threshold signature is valid. In case it is valid, it relays the message, obtains the hash value and the signature as proof for the hash, and stops participating. On the other hand, the View Synchronizer keeps bringing the processes on the same view till the leader of the view is correct. The module keeps the processes to the same view for time that is sufficient to reach a decision. The cooperation of the two modules is crucial for the correctness of our vector dissemination algorithm.

Roadmap. The rest of this work is organized as follows. Section 1 is the introduction, section 2 is the related work, section 3 is the preliminary material, section 4 presents the structure of our consensus algorithm, section 5 presents the vector dissemination module, and section 6 is the conclusion.

2 Related Work

In this section, we discuss the related work in the field of Byzantine (vector) consensus protocols.

RareSync. RareSync [1] is a View Synchronizer with $O(f)$ latency and $O(n^2)$ worst-case communication complexity. It is designed to solve view synchronization, which is the problem of bringing the processes in the same view with a correct leader for a sufficient amount of time. RareSync achieves such results by grouping views into epochs, with $f+1$ views per epoch, and manages to accomplish view synchronization in a constant number of epochs after GST, which is the time when the message delays become bounded by a known time δ.

The main innovation of RareSync is that processes rely on local clocks to proceed over views, and they only communicate at the end of an epoch. When a process finishes the last view of an epoch, it informs the rest of the system by broadcasting an epoch completion ($EPOCH\text{-}COMPLETED$) message. Upon receiving $n-f$ such messages from other processes for the same epoch, and upon waiting δ time, each process enters the new epoch and informs all the other processes by broadcasting an enter epoch ($ENTER\text{-}EPOCH$) message. When processes that are currently in an epoch e deliver such a message for an epoch $e' > e$, they wait for δ time and then they also enter epoch e'. The reason for this δ time wait is to allow processes to receive possible broadcasts for more advanced epochs.

RareSync analysis proves that this method allows the processes will synchronize for at least Δ time in every view of the first epoch entered after GST. This implies that by using RareSync, synchronization can be achieved in a constant number of epochs (and views) after GST.

QUAD. QUAD [1] is a partially synchronous Byzantine consensus protocol that exposes the following interface:

1. Request: propose(Value u). Proposes a value u for consensus.
2. Indication: decide(Value v). Decides a value v from consensus.

The protocol was initially proposed for Byzantine consensus, where the messages contain single proposals and have constant size, $O(1)$. In such settings, QUAD is able to achieve consensus in $O(n^2)$, with linear latency of $O(f)$. In addition, QUAD can be adapted for Byzantine vector consensus, but unfortunately the linear size of messages that contain $n-f$ values, $O(n)$ result in protocols that achieve consensus in $O(n^3)$.

Authenticated vector consensus. The Authenticated vector consensus approach [2] is a straightforward adaptation of QUAD for proposals that contain vectors, achieving $O(n^3)$ communication complexity and

$O(f)$ latency. Every process broadcasts its value and waits until it delivers $n-f$ proposals from the other system processes. Then, an input configuration vector $v \in \Re^{(n-f)}$ and a storage proof Σ are constructed from the proposals.

The pair (v,Σ) is proposed to QUAD by each process. As QUAD is a distributed protocol for Byzantine consensus, it responds by deciding one pair of v and Σ. The decided vector of QUAD is the final decision of the Authenticated vector consensus algorithm. In this protocol, the messages proposed to QUAD for consensus have $O(n)$ size, as they contain the proposal vector of $n-f$ values and the proof. In order to reach a decision, the total communication complexity of QUAD is $O(n^3)$, with a latency of $O(f)$, as the message size does not affect how fast a decision is reached.

Vector dissemination consensus. The cubic communication complexity of Authenticated vector consensus can be reduced by exploiting a Slow Broadcast-Based vector dissemination. The algorithm achieves $O(n^2 \log n)$ communication complexity, but it is impractical due to the latency of $O(n^f)$. Instead of proposing vectors to QUAD, the processes broadcast their values to the system. When a process gathers $n-f$ proposals, it creates an input configuration vector and initiates the Vector Dissemination protocol. Each correct process eventually obtains a hash value H and a storage proof sp for a vector. Then, the processes propose the pair (H,sp) to QUAD, which is a message of $O(1)$ size. Thus, QUAD responds with a selected pair. Having obtained the selected hash value of the decided vector, which is the same for each correct process due to the agreement property of consensus, the last part of the algorithm initiates the ADD protocol [3], an $O(n^2 \log n)$ protocol for asynchronous data dissemination that ensures that every process will obtain the vector that corresponds to the decided hash value.

The communication complexity of the algorithm is the sum of the communication complexities of the initial broadcast, the Vector Dissemination, QUAD, and ADD, which result in $n \cdot O(n) + O(n^2) + O(n^2) + O(n^2 \log n) = O(n^2 \log n)$. Due to the usage of Vector Dissemination, the latency of the algorithm is $O(n^f)$. A concrete analysis of the algorithm, along with proofs and pseudocode is presented in [2].

3 Preliminaries

In this section, we present the system model and preliminary material.

3.1 System

Processes. Our system consists of a static set of n processes, which we denote as $\{P_1, P_2, ..., P_n\}$. Our Byzantine distributed context assumes that there might be up to f faulty processes, where $n = 3f + 1$. Faulty processes can behave in an arbitrary way. For any process that is not faulty, we say that it is correct. Processes may communicate through authenticated point-to-point links, and the communication between two processes is reliable: If a correct process sends a message to another correct process, the message is eventually received. We assume that processes have local hardware clocks. Furthermore, we assume that local steps of processes take zero time, as the time needed for local computation is negligible compared to message delays.

Partial Synchrony. We consider a partially synchronous model as introduced in [5]. For every execution of an algorithm in such a model, there exists a Global Stabilization Time (GST), which is unknown to the processes. Before GST, message delays are unbounded and local clocks may drift arbitrarily, but after GST message delays are bounded by a known delay δ, and the clocks do not drift.

3.2 Cryptographic Primitives

We assume a (k,n)-threshold signature scheme [7], where $k = 2f+1 = n-f$. In this scheme, each process holds a distinct private key and there is a single public key. Each process P_i can use its private key to produce a partial signature of a message m by invoking $ShareSign(m)$. A partial signature $tsignature$ of a message m produced by a process P_i can be verified by $ShareVerify(m, tsignature)$. Finally, set $S = \{tsignature\}$ of partial signatures, where $|S| = k$ and, for each $tsignature_i \in S$, $tsignature = ShareSign(m)$, can be combined into a single (threshold) signature by invoking $Combine(S)$. The produced threshold signature has the same size as the partial ones. A Byzantine process cannot produce a valid threshold signature with less than $k = n-f = 2f+1$ partial signatures. We suppose that we have a provided *valid_SP* function, that given a threshold signature, responds $\{true, false\}$ based on if the signature is valid or not.

3.3 View Synchronization

Problem Definition. The view synchronization problem is the problem of bringing all processes in the same view with a correct leader for a sufficiently long time. The views are denoted as a set $Views = \{1,2,3,...\}$, and for each view, $u \in Views$ we have a designated leader process denoted as $leader(u)$. View Synchronizers are responsible for advancing the processes through views after keeping them in a view for sufficiently long, till they reach a view with a correct leader.

Interface. Thus, they expose the following interface:

- Indication: advance(View u). The process advances to view u.

We say that a process P enters a view u at time t only if the advance indication occurred at time t. If another advance indication occurs for another view u' at time t', then we say that P remained in view u during $[t,t')$. If no newer advance occurs, then P will remain in view u.

Synchronization time. We now define the notion of Synchronization time, which is the time when all correct processes overlap in the same view for at least Δ time.

Definition 1 (Synchronization time)**.** *Time t_s is a synchronization time if (1) all correct processes are in the same view from time t_s to (at least) time $t_s + \Delta$ and (2) the leader of the view is correct.*

View synchronization ensures the eventual synchronization property which states that there exists a synchronization time at or after GST.

Complexity. Let Synchronizer be a partially synchronous view synchronizer and let $\varepsilon(Synchronizer)$ denote the set of all possible executions. Let $a \in \varepsilon(Synchronizer)$ be an execution and $t_s(a)$ be the first synchronization time at or after GST in $a(t_s(a) \geq GST)$. We define the communication complexity of a as the number of words sent in messages by all correct processes during the time period $[GST, t_s(a)+\Delta]$. The latency complexity of a is $t_s(a)+\Delta-GST$. The communication complexity of Synchronizer is denoted as

$$\max_{a \in \varepsilon(Synchronizer)} \left\{ \text{Communication Complexity of } a \right\} \tag{1}$$

And the latency as

$$\max_{a \in \varepsilon(Synchronizer)} \left\{ \text{Latency of } a \right\} \tag{2}$$

3.4 Vector Dissemination

Problem Definition. In vector dissemination, every process proposes its vector of $n-f$ values, and eventually every correct process obtains a hash value and a proof of a vector.

Interface. The module exposes the following interface:

- Request: disseminate(Vector *vec*). Disseminates vector *vec* to the system
- Indication: obtain(Hash_Value H, Storage_Proof sp). Obtains a hash value and a storage proof for a disseminated vector

Properties. Formally, vector dissemination has the following properties:

1. Termination: Every correct process eventually obtains a hash value and a storage proof.
2. Integrity: If a process obtains a hash value H and a storage proof sp, then valid_SP(H,sp)=true, i.e. sp is a valid signature for the disseminated vector.
3. Redundancy: Let a process obtain a storage proof sp which for some hash value H, valid_-SP(H,sp)=true. Then, at least $f+1$ correct processes have cached a vector *vec*, such that $hash(vec) = h$.
4. δ-Closeness: Let t_{first} be the first time a correct process obtains a hash value and a storage proof. Then, every correct process obtains a hash value and a storage proof by time $max(GST, t_{first}) + \delta$.

3.5 Vector Consensus

Problem Definition. In distributed systems, consensus refers to the setting where processes propose an initial value and eventually decide on the same value. Vector consensus allows the processes to agree on a vector of $n-f$ values.

Interface. Vector consensus exposes the interface below:

- Request: propose(Value u). Proposes a scalar value u for consensus.
- Indication: decide(Vector v). Decides a vector v of $n-f$ values.

Properties. Vector consensus exposes the following properties:

1. Agreement: No two correct processes decide differently.
2. Termination: Every correct process eventually decides.
3. Integrity: No process proposes more than once.
4. Vector Validity: Let a correct process decide a vector v of $n-f$ values. Then, every value u of v was proposed by a process P to consensus.

Complexity. Let Consensus be an instance of a leader-based partially synchronous Byzantine consensus protocol and let $\varepsilon(Consensus)$ be the set of all possible executions of the protocol. Let $a \in \varepsilon(Consensus)$ be execution and $t_d(a)$ be the first time that all processes decide a vector in a. A word contains a constant number of signatures and values. Each message contains at least a single word. We denote

the communication complexity of a as the number of words sent in messages by all correct processes during the time period $[GST, t_d(a)]$, if $GST > t_d(a)$, the communication complexity of a is 0. The latency complexity of a is $max(0, t_d(a) - GST)$. Thus, the communication complexity of the algorithm is denoted as

$$\max_{a \in \varepsilon(Consensus)} \left\{ \text{Communication Complexity of } a \right\}$$

And the latency as

$$\max_{a \in \varepsilon(Consensus)} \left\{ \text{Latency of } a \right\}$$

Note that the communication complexity and latency contributed by the Byzantine processes is not considered.

4 Structure

In this section, we present the structure of our Byzantine consensus protocol. During this section, we use our proposed leader-based vector dissemination protocol to present our vector consensus algorithms, and we explain its details in section 5.

4.1 Overview

Our Byzantine consensus protocol consists of several modules that ensure the correctness of the algorithm. The usage of each module represents a specific phase of the algorithm, as we explain below:

Proposal Broadcasting. Each process starts executing the protocol by broadcasting its proposal through a best-effort broadcast primitive, which ensures that if the process that broadcasts the proposal is correct, every correct process will eventually deliver the proposal. The processes then wait till they deliver $n - f$ distinct proposals.

Vector Dissemination. Upon delivering the $n - f$ proposals, each process composes a vector containing those values, namely the input configuration vector, and initiates our leader-based vector dissemination module with this vector. The module eventually responds with a Hash_Value H and a Storage_Proof sp of a vector. A complete description and analysis of the module are presented in section 5.

Agreement. Upon obtaining the pair of Hash_Value and Storage_Proof, the processes need to agree on a pair. To do so, each process proposes its pair to QUAD and eventually processes reach a consensus. Due to the agreement property of consensus, each process decides the same value of Hash_Value and Storage_Proof. Proposing a message containing a pair of values has constant size $O(1)$, which is an important feature for keeping the complexity of QUAD to $O(n^2)$.

Reconstruction. Now that each process has the same Hash_Value for the decided vector, the original vector should be reconstructed. As we will show in section 5, some processes may have cached a vector that has the corresponding hash value, but some other processes may have not, due to the redundancy property of vector dissemination. For this reason, each process initiates a module for asynchronous data dissemination, namely ADD [3], which eventually responds with the reconstructed vector. When each process obtains the final vector from ADD, it decides the vector and consensus is complete.

4.2 Protocol

The pseudocode for our vector consensus protocol is presented in algorithm 1, from the perspective of process P_i. It starts by broadcasting its proposal for the rest of the system. Upon delivering $n-f$ proposals from different processes, it composes an input configuration vector and initiates vector dissemination. Upon obtaining a Hash_Value and a Storage_Proof it proposes the pair to QUAD, which allows the processes to reach a consensus on the Hash_Value of the decided vector. The last part of the algorithm initiates the ADD module to reconstruct the decided vector.

Algorithm 1 Vector consensus pseudocode for process P_i

1: **Uses**:
2: Best-effort broadcast, instance *beb*
3: Vector dissemination, instance *dissemination*, of section 5, with $|VE|$ views per epoch.
4: QUAD, instance *quad* [1]
5: ADD, instance *add* [3]
6: **upon** init:
7: Integer $received\_proposals \leftarrow 0$
8: Map(Process $\rightarrow$ Value) $proposals_i \leftarrow empty$
9: Map(Process $\rightarrow$ Message) $messages_i \leftarrow empty$
10: **upon** propose(Value u):
11: Signature $sig_i \leftarrow$ signature of message $\{PROPOSAL, u\}$
12: **invoke** $beb.broadcast(sig_i)$
13: **upon** reception of sig_j of message $\{PROPOSAL, u_j\}$ from P_j and $received\_proposals_i < n-f$:
14: $received\_proposals_i \leftarrow received\_proposals_i + 1$
15: $proposals_i[P_j] \leftarrow u_j$
16: $messages_i[P_j] \leftarrow sig_j$
17: **if** $received\_proposals = n-f$:
18: Vector $vec \leftarrow$ Create an input configuration vector from the $proposals_i$
19: $disseminator.disseminate(vec)$
20: **upon** $disseminator.obtain$(Hash_Value H, Storage_Proof sp):
21: **if** not yet proposed to *quad*:
22: **invoke** $quad.propose((H, sp))$
23: **upon** $quad.decide$((Hash_Value H', Storage_Proof sp')):
24: Vector $vec' \leftarrow$ a cached vector with H' hash value
25: **invoke** add.input(vec')
26: **upon** $add.output$(Vector vec''):
27: **trigger** $decide(vec'')$

4.3 Correctness

Here, we discuss the correctness of our protocol. The complete correctness proofs are presented in [2], as our algorithm is an adaption of the consensus algorithm, using our leader-based vector dissemination module.

Lemma 4.1. *Algorithm 1 satisfies Agreement.*

Proof. From the presented pseudocode, every process will gather (at least) $n-f$ proposals and it will create a vector of $n-f$ values. This vector is used to initiate the vector dissemination module. Due to the termination property of vector dissemination, each correct process will eventually obtain a pair of Hash_Value and Storage_Proof and it will propose it to QUAD. Due to the termination and agreement properties of QUAD, each correct process will eventually decide on a hash value of a vector, and this hash will be the same for all correct processes participating in consensus. Then, the vector is reconstructed correctly due to the asynchronous data dissemination of ADD (for more information refer to [3]) and because at least $f+1$ processes will have cached a vector with the corresponding hash value due to the Redundancy property of vector dissemination. Thus, every correct process will reconstruct and decide the same vector. □

Lemma 4.2. *Algorithm 1 satisfies Termination.*

Proof. The termination property of the algorithm comes from the fact that every correct process will eventually receive $n-f$ proposals to initiate vector dissemination and from the termination properties of the other modules used in the algorithm. □

Lemma 4.3. *Algorithm 1 satisfies Integrity.*

Proof. Integrity is satisfied from the fact that each process obtains a single pair of Hash_Value and Storage_Proof, because as we present in section 5 each process stops participating in the algorithm when it obtains a pair for the first time, and from the integrity property of QUAD consensus. □

Lemma 4.4. *Algorithm 1 satisfies Vector Validity.*

Proof. This is a direct consequence of the fact that every correct process initializes a vector of $n-f$ proposals from distinct processes in order to begin vector dissemination.

□

Thus, based on the above, we have the following theorem, regarding the correctness of the algorithm.

Theorem 4.5. *Algorithm 1 is correct.*

4.4 Complexity

Now, we prove the communication complexity and latency of the protocol, when we use $|VE|$ views per epoch for the vector dissemination module. The communication complexity of vector dissemination is $O(\lceil \frac{(f+1)+|VE|^2}{|VE|} \rceil)$ and the latency is $O(view\_duration \cdot (|VE|+(f+1)))$. The detailed explanation and proof are presented in section 5. The communication complexity of QUAD is $O(n^2)$ and the latency is $O(f)$. Also, communication complexity of the asynchronous ADD module is $O(n^2 \log n)$.

Communication Complexity. The communication complexity of the protocol can be modeled as follows

$$O(n^2) + O(\lceil \frac{(f+1)+|VE|^2}{|VE|} \rceil \cdot n^2) + O(n^2) + O(n^2 \log n)$$

because it is computed as the sum of the communication complexity contributed by each module of the protocol. Thus, we have the following theorem.

Theorem 4.6. *For $|VE| = \sqrt{n}$, the communication complexity of vector consensus is $O(n^2\sqrt{n})$*

Proof. The communication complexity of algoritm 1 for $|VE| = \sqrt{n}$ is

$$O(n^2) + O(n^2\sqrt{n}) + O(n^2) + O(n^2 \log n) = O(n^2\sqrt{n})$$

□

Latency. Now we present the latency. The latency of the protocol is the sum of the latency of each module, namely the broadcasting, the leader-based vector dissemination, and QUAD (ADD is asynchronous). Thus, the latency can be modeled as

$$O(1) + O(view\_duration \cdot (|VE| + (f+1))) + O(f)$$

Thus, we have the following theorem about the final latency of consensus.

Theorem 4.7. *For $|VE| = \sqrt{n}$, the latency of vector consensus is $O(n\sqrt{n})$*

Proof. The latency of the algorithm for $|VE| = \sqrt{n}$, based also on section 5 is:

$$O(1) + O(n\sqrt{n}) + O(f) = O(n\sqrt{n})$$

because in this setting, as we show in the next section, the view duration of the leader-based vector dissemination is $O(\sqrt{n})$. □

Value of $|VE|$. The proofs above exposed the crucial role that views per epoch, $|VE|$ play in the final latency and complexity of vector consensus. By following the same chain of thought, it can be proven that using $|VE| = f+1$ the communication complexity of vector consensus is $O(n^3)$ and the latency is $O(f)$. In general, the value of $|VE|$ exposes a trade-off between the communication complexity and latency of the resulting algorithm, as more views in an epoch result in better latency, as a view with a correct leader can be reached in only one epoch, but it requires more communication complexity. Our research in this work focuses on reducing the communication complexity of Byzantine vector consensus, and that is why we proceeded with $|VE| = \sqrt{n}$, which significantly reduces the communication complexity of the protocol. To the best of our knowledge, our approach is the only Byzantine vector consensus algorithm in a partial synchronous setting that achieves sub-cubic communication complexity and sub-quadratic latency.

5 Vector Dissemination

In this section, we present our leader-based vector dissemination protocol, which we used in the previous section to describe our vector consensus algorithm. The protocol operates in views, where in each view a designated leader tries to disseminate its vector to the other processes. The views are organized into epochs, and each epoch contains $|VE|$ views. The protocol relies on two modules, the View Core, responsible for the actions of the processes within views, and the View Synchronizer, responsible to advance processes between views in order to bring the processes in view with a correct leader for a sufficiently long time.

5.1 Protocol

The pseudocode of the leader-based vector dissemination protocol is presented in algorithm 2, from the perspective of a process P_i. The protocol starts when a vector dissemination request triggers the dissemination event of the algorithm, which subsequently initializes the core and synchronizer modules. Throughout the procedure, the synchronizer triggers advance events, that inform the core to start executing the actions corresponding to this view. Eventually, a view with the correct leader will be reached, and the processes will remain there for sufficient time to obtain a hash. When they obtain a Hash_Value, the core triggers the obtain event, which indicates that a Hash_Value H and a Storage_Proof sp of a vector have been retrieved. We explain both our View Core and the View Synchronizer (which is a Generalized version of RareSync of section 2) in the next parts of this section.

Algorithm 2 Leader-based vector dissemination pseudocode for process P_i

```
1: Modules
2:     View_Core core ← View Core with batch size |VE|
3:     View_Synchronizer synchronizer ← Generalized RareSync with |VE| epochs
4: upon disseminate(Vector v):
5:     core.init(v)
6:     synchronizer.init
7: upon synchronizer.advance(View u):
8:     core.start_executing(u)
9: upon core.obtain(Hash_Value H, Storage_Proof sp):
10:    trigger obtain(H,sp)
```

5.2 View Core

The view core is responsible for the actions of the processes during a view. For a specific view $v \in Views$, where $Views = \{1, 2, ..., |VE|\}$, the leader process of v, denoted as $leader(v)$ tries to disseminate its vector to the rest of the system. It starts by storing the hash H_i of its vector vec and then it broadcasts the vector to the rest of the system in a $PROPOSE$ message. The leader broadcasts the message to $|VE|$ processes at a time (batch size), and then it waits δ time before broadcasting to the next batch of processes. Note that we choose batch size $|VE|$ for simplicity and concreteness for our analysis, as in the general case, this can be an arbitrary number. Batch size usage is crucial to keep the communication complexity quadratic, as we will later show in our complexity analysis.

Every process P_j that delivers the message containing the vector vec from the leader, caches vec and creates a signature containing the hash H_i of the vector, and sends it back to the leader to inform that the vector is cached, using a $STORED$ message.

Upon receiving $n - f$ messages that other processes have cached the vector, the leader combines the messages in a storage proof sp and broadcasts a $DECIDE$ message containing the hash H_i of the vector along with sp. Any process that delivers the $DECIDE$ message verifies that sp is a valid threshold signature. If it is, it triggers obtain with H_i and sp, rebroadcasts the $DECIDE$ message, and stops participating in the dissemination. We present the pseudocode of the protocol in algorithm 3, from the perspective of a process P_i and a specific view u.

Algorithm 3 View Core pseudocode for view u and process P_i

1: **Uses**
2: Best-Effort Broadcast, instance *beb*
3: **upon** init(Vector v):
4: Map (Hash_Value $\rightarrow$ Vector) $vectors_i \leftarrow$ empty
5: Set (Process) $disseminated_i \leftarrow$ empty
6: Hash_Value $H_i \leftarrow None$
7: **if** process P_i is $leader(u)$
8: $H_i \leftarrow hash(v)$
9: **while** vector v is not broadcasted to all processes
10: **invoke** $beb.broadcast(\{PROPOSE, v\})$ to $|VE|$ processes
11: **wait** δ time
12: **upon** $beb.deliver$(Message $\{PROPOSE, v'\}$, Process P_j):
13: **if** process $P_j \notin disseminated_i$
14: $disseminated_i \leftarrow disseminated_i \cup \{P_j\}$
15: $vectors_i[hash(v')] \leftarrow v'$
16: Signature $sig \leftarrow$ Signature of the message $\{STORED, hash(v')\}$
17: **send** sig to P_j
18: **upon** reception of sig of message $\{STORED, H'\}$ such that $H' = H_i$ from $n - f$ distinct processes:
19: Storage_Proof $sp \leftarrow Combine(\{sig | sig$ is a signature of a received STORED message$\})$
20: **invoke** $beb.broadcast(\{DECIDE, H_i, sp\})$
21: **upon** reception of message $\{DECIDE, H', sp'\}$:
22: **if** sp' is a valid $n - f$ threshold signature
23: **invoke** $beb.broadcast(\{DECIDE, H', sp'\})$
24: **trigger** $obtain(H', sp')$
25: **stop participating** in vector dissemination

5.3 View Synchronizer

For our View Synchronizer, which is responsible for bringing the processes to the same view with a correct leader for a sufficiently long time, we use a generalized version of RareSync, described in section 2, in order to work with an arbitrary number of views per epoch, $|VE|$. We call our new version Generalized RareSync, and we describe the protocol with its complexity here. Views in Generalized RareSync are organized into epochs, processes advance in views based on local clocks and communicate by broadcasting at the end of each epoch. The main modification of the Generalized version is that it is not necessary to have $f+1$ views in each epoch, but each epoch consists of $|VE|$ views. Such generalization is important, because it allows us to tune the communication complexity and latency according to the needed task, and because it exposes the important trade-off between latency and complexity in our Generalized version of RareSync. Although the protocol works the same way as the original one, changing the number of views in an epoch has a crucial impact on latency and communication complexity because depending on the value of $|VE|$ an epoch with processes synchronized in each view may not be sufficient. This is the case where $|VE| < f+1$ as in the worst case, no view of the epoch will have a correct leader. For the rest of the section, we explore this impact on the complexity of Generalized RareSync, based on the value of $|VE|$.

Complexity. A correct process enters an epoch e at time t if and only if the process enters the first view of e at time t. We denote by t_e the first time a correct process enters epoch e.

Through a series of results, some of them derived directly from RareSync, we prove that the latency of Generalized RareSync is $O(view\_duration \cdot (|VE| + (f+1)))$. We start by defining two important epochs, e_{max} and e_{final}.

Definition 2 (Epoch e_{max})**.** *Epoch e_{max} is the last epoch entered by the most advanced process before GST.*

Definition 3 (Epoch e_{final})**.** *Epoch e_{final} is the first epoch entered by the most advanced process at or after GST.*

Then, we proceed with our results. Many of the results are direct consequences of the original RareSync algorithm, but we give a brief proof sketch for all of them. For extended explanations and proofs about those results, refer to [1].

Lemma 5.1. *For any epoch $e \geq e_{final}$, no correct process broadcasts an EPOCH-COMPLETED message for e before time $t_e + epoch\_duration$, where $epoch\_duration = |VE| \cdot view\_duration$.*

Proof. This statement is a direct consequence of the fact that after *GST*, where local clocks do not drift, a process needs exactly $epoch\_duration$ time to go through all $|VE|$ views of the epoch. Remember that before *GST*, there can be arbitrary drifts between the clocks of different processes. □

Lemma 5.2. *Any correct process enters epoch e_{max} or any greater epoch by $GST + 3\delta$.*

Proof. Any correct process that is in an epoch older than e_{max} by the time *GST* occurs, will receive the *ENTER-EPOCH* message for a newer epoch e by time $GST + \delta$ and it will wait δ time. By the end of this time, it is possible that it will receive another *ENTER-EPOCH* message for a newer epoch $e' > e$, and it will wait another δ time, until finally entering the epoch by $GST + 3\delta$. □

Lemma 5.3. *Every correct process enters epoch e_{final} by time $t_{efinal} + 2\delta$.*

Proof. Let t_{efinal} be the time when the most advanced process enters epoch e_{final} and broadcasts the *ENTER-EPOCH* message. As $t_{efinal} \geq GST$ (from definition 3), every process receives the message by $t_{efinal} + \delta$ at the latest. After receiving the message, processes wait δ time before entering e_{final}, and thus, all processes enter epoch e_{final} by $t_{efinal} + 2\delta$. □

Lemma 5.4. *In every view of e_{final} processes overlap in a view for at least Δ time.*

Proof. No future epoch can be entered before time $t_{efinal} + epoch\_duration$. This is precisely enough time for the first correct process (the one to enter e_{final} at t_{efinal}) to go through all $|VE|$ views of e_{final}, spending *view_duration* time in each view. Since clocks do not drift after *GST* and processes spend the same amount of time in each view, the maximum delay between processes is 2δ. This is a direct consequence of the previous result, as every process will enter the first view of this epoch by $t_{efinal} + 2\delta$.

Thus, all correct processes overlap with other for at least $view\_duration - 2\delta = \Delta$ time, for every view of e_{final}. □

Lemma 5.5. *For every view of every epoch $e > e_{final}$, processes overlap in a view for at least Δ time.*

Proof. This result is a direct consequence of the previous theorem. As after *GST* local clocks do not drift, and at e_{final} the processes are synchronized for every view, they will still be synchronized for every view that will be entered for any epoch $e > e_{final}$. □

Lemma 5.6. *Let t_s be the synchronization time and t_{efinal} be the time when all correct processes enter the epoch e_{final}. Then, $t_s \leq t_{efinal} + (f+1) \cdot view\_duration - \Delta$*

Proof. Based on the results presented above,

$$t_s \leq t_{efinal} + e_n \cdot epoch\_duration - \Delta \Rightarrow$$

$$t_s \leq t_{efinal} + \lceil \frac{f+1}{|VE|} \rceil \cdot |VE| \cdot view\_duration - \Delta \Rightarrow$$

$$t_s \leq t_{efinal} + (f+1) \cdot view\_duration - \Delta$$

□

Lemma 5.7. *Let t_{efinal} be the time when all correct processes enter the epoch e_{final}. Then, $t_{efinal} \leq GST + epoch\_duration + 4\delta$*

Proof. This comes from the following observations.

1. $GST + \delta$: Every process receives *ENTER-EPOCH* message for epoch e_{max}
2. $GST + 2\delta$: Every process enters first view of epoch e_{max}, after waiting δ time.
3. $GST + 2\delta + epoch\_duration$: Every process completes the epoch and broadcasts the *EPOCH-COMPLETED* message.
4. $GST + 2\delta + epoch\_duration + \delta$: Every process receives $2f+1$ *EPOCH-COMPLETED* messages.
5. $GST + 2\delta + epoch\_duration + \delta + \delta$: Every process enters epoch e_{final}, after waiting δ time.

Thus,

$$t_{efinal} \leq GST + epoch\_duration + 4\delta$$

□

Latency. The latency of Generalized RareSync is $O(view\_duration \cdot (|VE| + f + 1))$

Proof. From the results above:

$$t_s \leq GST + epoch\_duration + 4\delta + (f+1) \cdot view\_duration - \Delta \Rightarrow$$

$$t_s - GST + \Delta \leq epoch\_duration + 4\delta + (f+1) \cdot view\_duration \Rightarrow$$

$$latency \leq epoch\_duration + 4\delta + (f+1) \cdot view\_duration \Rightarrow$$

$$latency \leq X \cdot view\_duration + 4\delta + (f+1) \cdot view\_duration$$

As 4δ is a constant, latency of the algorithm is

$$latency = O(|VE| \cdot view\_duration + (f+1) \cdot view\_duration) \Rightarrow$$

$$latency = O(view\_duration \cdot (|VE| + f + 1))$$

We should note that the *view_duration* is not constant. We later provide more information about how the *view_duration* is defined based on the batch size that the View Core uses.

□

We now proceed with the communication complexity. Processes running Generalized RareSync need only one communication round per epoch, where all processes broadcast their *EPOCH-COMPLETED* message. Thus the per epoch communication complexity of Generalized RareSync is $n \cdot O(n) = O(n^2)$.

Communication Complexity. The communication complexity of Generalized RareSync is $O(\lceil \frac{f+1}{|VE|} \rceil \cdot n^2)$.

Proof. As processes need to traverse over $(f+1)$ views in total, the total number of epochs needed to reach synchronization is 1 + $\lceil \frac{f+1}{|VE|} \rceil$, including the first epoch on *GST*, that the processes will be unsynchronized in the worst case. As each epoch requires quadratic communication complexity, the overall communication complexity of the algorithm is

$$O(\lceil \frac{f+1}{|VE|} \rceil \cdot n^2)$$

□

Theorem 5.8. *Generalized RareSync with $|VE|$ views per epoch has a latency of $O(view\_duration \cdot (|VE| + (f+1)))$ and communication complexity of $O(\lceil \frac{f+1}{|VE|} \rceil \cdot n^2)$.*

The proof of correctness is easily derived from the proof of the original RareSync, presented in [1].

5.4 Correctness

Now we prove the correctness of leader-based vector dissemination, implemented by algorithm 2

Lemma 5.9. *Algorithm 2 satisfies Termination.*

Proof. As we showed for Generalized RareSync, the processes will overlap in each view of any epoch $e \geq e_{final}$, and they will be passing over views till they reach a view where the leader is correct so they can synchronize. Let $view_{sync}$ be the first view of any epoch $e \geq e_{final}$ where $leader(view_{sync})$ is correct.

Let t_{final} be the time where $leader(view_{sync})$ broadcasts the vector to all processes (this will be $O(\frac{n}{|VE|})$ time after the beginning of the epoch).

By the time $t_{final} + \delta$, every correct process receives it and sends back the signed message. The leader receives all acknowledgment messages by $t_{final} + 2\delta$, composes the storage proof, and broadcasts again. By $t_{final} + 3\delta$ all correct processes have obtained a hash value and storage proof, ensuring termination. □

Lemma 5.10. *Algorithm 2 satisfies Integrity.*

Proof. This is a direct consequence of the fact that the algorithm checks if sp is valid before triggering $obtain$. □

Lemma 5.11. *Algorithm 2 satisfies Redundancy.*

Proof. Let a process, either correct or faulty, obtain a storage proof sp and a hash value H, which $valid\_SP(H, sp) = true$ (we suppose that *valid_SP* is provided and correctly verifies that sp is valid). This means that $n - f$ processes signed the message containing the hash value of the stored vector. Among these $n - f$ processes, at least $n - f - f = 3f + 1 - 2f = f + 1$ are correct. As seen in the algorithm, each one of those $f + 1$ correct processes caches its vector before communicating with the leader, satisfying redundancy. □

Lemma 5.12. *Algorithm 2 satisfies δ-Closeness.*

Proof. Let P_{first} be a correct process that obtains a hash value and a storage proof at time t_{first}. Before P_{first} stops participating in the protocol, it rebroadcasts the hash value it obtains and the corresponding storage proof. Thus, by time $max(GST, t_{first} + \delta)$, every process receives the storage proof and the hash value from P_{first}. □

Theorem 5.13. *Algorithm 2 is correct.*

5.5 Complexity

Here, we analyze the complexity of our leader-based vector dissemination. The complexity is calculated by the partial complexities of View Core and View Synchronizer, while the latency is defined by the latency of the View Synchronizer. We begin by presenting some intermediary results which will be used later to prove the overall complexity of the algorithm.

Lemma 5.14. *Let $view_{sync}$ be the view where the processes finally synchronize, i.e. they overlap in $view_{sync}$ for a sufficiently long time, and $leader(view_{sync})$ is correct. Then, the communication complexity of the View Core for this view is $O(n^2)$.*

Proof. In a $view_{sync}$ where $leader(view_{sync})$ is correct, it disseminates its vector $v \in \Re$ by broadcasting. Then, the processes respond by sending their acknowledgments to the leader, which then broadcasts again the final decision containing the Hash_Value of the decided vector and the Storage_Proof, which is of constant size. When every process receives the final decision message, it relays it to the system.

The above observations result in communication complexity of:

$$O(n) \cdot O(n) + n \cdot O(1) + O(1) \cdot O(n) + n \cdot O(n) = O(n^2)$$

□

Lemma 5.15. *Let $view_{faulty}$ be any view where the $leader(view_{faulty})$ is faulty, and the processes are synchronized. Then, the communication complexity of any such view is $O(n)$.*

Proof. In any view $view_{faulty}$ that the leader $leader(view_{faulty})$ is Byzantine, the system cannot reach a decision, i.e. obtain the Hash_Value and Storage_Proof of the leader. Since we have a faulty leader, we do not count the communication complexity of the messages emitted by any Byzantine process during this view. Thus, the communication complexity of the view comes from the acknowledgment messages that the correct processes will send back to the leader, ensuring that they received the message. The communication complexity for such views is:

$$(n-f) \cdot O(1) = O(n)$$

□

Lemma 5.16. *The communication complexity of the View Core of leader-based vector dissemination for $|VE|$ epochs per view is $O(|VE| \cdot n^2)$.*

Proof. When *GST* occurs, the most advanced process traverses over a view of e_{max}, as described by Generalized RareSync. In the worst case, when *GST* occurs, each correct process has entered a different view of an epoch and is the leader of this epoch, ready to broadcast its vector. This scenario is the worst case because the processes are completely synchronized, and they will remain in their views at most by $GST + 3\delta$, as we presented in Generalized RareSync. In this time interval, they will act as leaders, and they will broadcast $3|VE|$ messages, because of the batch broadcasting that the View Core implements. This means that before joining e_{max} or any greater epoch, the communication complexity is

$$O(n) \cdot O(|VE|) \cdot O(n) = O(|VE| \cdot n^2)$$

because $O(n)$ leaders will broadcast $O(|VE|)$ messages of $O(n)$ size. Then, in the worst case, the processes will enter e_{max}. This case can be described as follows: In that epoch, each view will have been entered by its correct leader process. This means that $|VE|$ leaders will concurrently broadcast their vectors to all processes, having communication complexity of

$$O(|VE|) \cdot O(n) \cdot O(n) = O(|VE| \cdot n^2)$$

because $O(|VE|)$ correct leaders will broadcast $O(n)$ messages of $O(n)$ size. The reason we cannot obtain a hash here, is that the processes are slightly misaligned.

After epoch e_{max}, processes enter e_{final}, where they overlap in each view of this epoch, and any epoch $e > e_{final}$. The processes need to pass over $f+1$ views, that correspond to $\lceil \frac{f+1}{|VE|} \rceil$ epochs in order to synchronize. In our setting, in the first f views the leader is Byzantine and in the last view, the leader

$\|VE\|$	Communication Complexity	Latency
$f+1$	$O(n^3)$	$O(f)$
$\sqrt{n}$	$O(n^2\sqrt{n})$	$O(n\sqrt{n})$

Table 1: Communication complexity and latency of leader-based vector dissemination for indicative values of $|VE|$.

is correct and the processes finally synchronize. The communication complexity of this procedure is $f \cdot O(n) + 1 \cdot O(n^2) = O(n^2)$. The overall communication complexity of the view core is:

$$|VE| \cdot O(n^2) + |VE| \cdot O(n^2) + f \cdot O(n) + 1 \cdot O(n^2) = |VE| \cdot O(n^2) + O(n^2) = |VE| \cdot O(n^2)$$

□

Theorem 5.17. *The communication complexity of leader-based vector dissemination with $|VE|$ views per epoch is $O(\lceil \frac{(f+1)+|VE|^2}{|VE|} \rceil \cdot n^2)$.*

Proof. We proved that the communication complexity of Generalized RareSync for $|VE|$ views per epoch is $O(\lceil \frac{f+1}{|VE|} \rceil \cdot n^2)$. The communication complexity of the View Core for $|VE|$ views per epoch is $O(|VE| \cdot n^2)$ Thus, the communication complexity of leader-based vector dissemination is

$$O(\lceil \frac{f+1}{|VE|} \rceil \cdot n^2) + O(|VE| \cdot n^2) = O(\lceil \frac{(f+1)+|VE|^2}{|VE|} \rceil \cdot n^2)$$

□

Theorem 5.18. *The latency of vector dissemination with $|VE|$ views per epoch is $O(\frac{n}{|VE|} \cdot (|VE| + (f+1)))$.*

Proof. The proof comes directly from the latency of Generalized RareSync, where the latency of vector dissemination with $|VE|$ views per epoch is $O(view\_duration \cdot (|VE| + (f+1)))$. Because the View Core broadcasts the proposal in batches of size $|VE|$, the overall view duration is $\frac{n}{|VE|}$. □

5.6 Discussion: Value of |VE|

Table 5.6 shows the latency and communication complexity for some indicative choices of VE. As we see, the choice of $|VE|$ exposes a trade-off between the communication complexity and the latency of the algorithm. If we use more views per epoch, the latency is linear, as we need fewer overall epochs to reach synchronization, but more views have a negative impact on communication complexity. Using fewer views per epoch has a negative impact on latency, which is not linear anymore, but the communication complexity is now sub-cubic.

6 Conclusion

This work presented a Byzantine vector consensus protocol that achieves vector consensus in sub-cubic communication complexity and sub-quadratic latency, thanks to the novel vector dissemination module it utilizes.

We started by presenting an overview of the vector consensus protocol, explaining how each module of the algorithm works, and then we proceeded by explicitly describing our leader-based vector dissemination protocol, which is crucial for the correctness and performance of the consensus algorithm. Our leader-based vector dissemination protocol utilizes a View Core and a View Synchronizer module, in order to correctly disseminate a vector. Our View Core works by enabling the leader process to start a "leader-to-all" and then "all-to-leader" communication policy while our View Synchronizer is a generalized version of RareSync, namely Generalized RareSync. Our generalization policy, which enables the module to work with an arbitrary number of views per epoch, has a significant role in the complexity and latency of the algorithm, based on the value of the views we select to use. Our consensus protocol achieves $O(n\sqrt{n})$ latency and $O(n^2\sqrt{n})$ communication complexity when the number of views per epoch is $\sqrt{n}$, $VE = \sqrt{n}$, and to the best of our knowledge is the only Byzantine vector consensus algorithm that can achieve both sub-cubic communication complexity and sub-quadratic latency in a partial synchronous environment of distributed processes.

Regarding future work, we plan to investigate more aspects of leader-based algorithms and research the possibility of even faster distributed protocols that achieve Byzantine vector consensus in a partially synchronous setting. Moreover, we plan to investigate the impact of our methods on different environment settings, in order to extend our work to synchronous environments.

Notes. This paper is part of an MSc. Research Project of EPFL. The related papers, reports, and presentations are available at GitHub.

References

[1] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic & Manuel José Ribeiro Vidigueira (2022): *Byzantine Consensus is Θ (n^ 2): The Dolev-Reischuk Bound is Tight even in Partial Synchrony!* Technical Report, Dagstuhl Publishing.

[2] Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic & Manuel Vidigueira (2023): *On the Validity of Consensus*. *arXiv preprint arXiv:2301.04920*.

[3] Sourav Das, Zhuolun Xiang & Ling Ren (2021): *Asynchronous Data Dissemination and its Applications*. Cryptology ePrint Archive, Paper 2021/777. Available at `https://eprint.iacr.org/2021/777`. `https://eprint.iacr.org/2021/777`.

[4] Danny Dolev & Rüdiger Reischuk (1985): *Bounds on Information Exchange for Byzantine Agreement*. *J. ACM* 32(1), p. 191–204, doi:10.1145/2455.214112. Available at `https://doi.org/10.1145/2455.214112`.

[5] Cynthia Dwork, Nancy A. Lynch & Larry J. Stockmeyer (1988): *Consensus in the presence of partial synchrony*. *J. ACM* 35, pp. 288–323.

[6] Leslie Lamport, Robert E. Shostak & Marshall C. Pease (1982): *The Byzantine Generals Problem*. *ACM Trans. Program. Lang. Syst.* 4, pp. 382–401.

[7] Benoît Libert, Marc Joye & Moti Yung (2016): *Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares*. *Theoretical Computer Science* 645,

pp. 1–24, doi:https://doi.org/10.1016/j.tcs.2016.02.031. Available at https://www.sciencedirect.com/science/article/pii/S0304397516001626.

[8] JongBeom Lim, Taeweon Suh, Joon-Min Gil & Heonchang Yu (2014): *Scalable and leaderless Byzantine consensus in cloud computing environments*. Information Systems Frontiers 16, pp. 19–34.

[9] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei & Chen Qijun (2017): *A review on consensus algorithm of blockchain*. In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2567–2572, doi:10.1109/SMC.2017.8123011.

[10] Nitin H. Vaidya & Vijay K. Garg (2013): *Byzantine Vector Consensus in Complete Graphs*. arXiv:1302.2543.

[11] Wei Yao, Junyi Ye, Renita Murimi & Guiling Wang (2021): *A Survey on Consortium Blockchain Consensus Mechanisms*. arXiv:2102.12058.